Títol: Paral·lelització del software de simulació PELE++ utilitzant GPUs

Autor: Xavier Oró Gay

Data: 13/06/2012

Director: Víctor Alejandro Gil Sepúlveda

Institució del director : BSC

Ponent: Rosa Maria Badia Sala

Departament del ponent: AC

Titulació: Enginyeria Informàtica

Centre: Facultat d'Informàtica de Barcelona (FIB)

Universitat: Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Paral·lelització del software de simulació PELE++ utilitzant GPUs

Xavier Oró Gay

- Juny 2012 -

Índex

1 Introducció

En el departament de Life Sciences del BSC utilitzen simulacions per computador per realitzar investigacions en el camp de les proteïnes. Aquestes simulacions requereixen molt de temps per dur-se a terme.

Un dels simuladors que utilitzen es diu PELE++ i esta sent desenvolupat en el propi departament. Per tal de millorar-ne el rendiment, i així reduir el temps necessari per realitzar les simulacions, els desenvolupadors es van plantejar explotar el sistemes multiprocessador actuals i paral·lelitzar el codi de les funcions més costoses. El sistema de paral·lelització plantejat, consisteix en usar els processadors de les targetes gràfiques actuals degut a la seva elevada potència de càlcul.

L'objectiu d'aquest projecte consistirà en transformar un grup de funcions del software de simulació PELE++ per ser executades en el processador de la targeta gràfica, i així alliberar la CPU d'aquests càlculs.

Per poder entendre les funcions ha paral·lelitzar, hem de conèixer el context i la temàtica del programa per saber el que fa. Per aquesta raó, la primera part del projecte consisteix en aprendre les nocions bàsiques de biologia relacionades amb les proteïnes, la dinàmica molecular i el funcionament de PELE++.

Quan ja coneguem el context i funcionament de PELE++, s'introduirà la tecnologia que volem utilitzar, en el nostre cas la paral·lelització en GPU, i els dos llenguatges que utilitzarem, CUDA i OpenCL.

Arribats a aquest punt, ja estarem preparats per començar la tasca de transformació del codi de les funcions, per poder trobar alguna estratègia que permeti la seva paral·lelització.

2 Les proteïnes

Les proteïnes són biomolècules (molècules que formen part de la matèria viva) formades per cadenes lineals d'aminoàcids.

En els éssers vius, les proteïnes realitzen diferents funcions i són de gran importància, ja que dirigeixen quasi tots els processos vitals. Algunes d'aquestes funcions són:

- Estructural: formen teixits i estructures com el col·lagen.
- Enzimàtica: augmenten la velocitat de les reaccions químiques, com la pepsina i la sacarasa.
- Hormonal: coordinar l'activitat conjunta de les cèl·lules com la insulina i hormona del creixement.
- Defensiva: el sistema immunitari utilitza proteïnes per identificar i neutralitzar elements estranys al cos, com ara bacteris, virus o paràsits. És el cas dels anticossos.
- Transport: es poden unir selectivament a molècules com l'hemoglobina amb l'oxigen o HDL i LDL amb el colesterol.
- Reguladores: regulen diferents funcions del nostre cos com la insulina que és una hormona polipeptídica.
- Contracció muscular: intervenen en el moviment muscular com la miosina.
- Funció homeostàtica: mantenen en diferents medis l'equilibri osmòtic com l'albumina.
- Energia: poden aportar fins a 4 kcal d'energia per gram en cas d'extrema necessitat.

2.1 Importància de les proteïnes i la seva investigació

El funcionament erroni de proteïnes pot derivar en malalties. Un error en la seqüencia d'aminoàcids pot ser-ne una de les causes.

La investigació en aquest camp pot permetre trobar remeis a malalties, com podria ser inserint els elements adequats a certes proteïnes. Un parell d'exemples de malalties relacionades amb les proteïnes són:

- L'alzheimer: provocat per un plegament incorrecte de proteïnes com l'APP.
- L'hipercolesterolèmia familiar: és deguda a la mutació genètica de la proteïna LDL que fa augmentar els nivells de colesterol a la sang.

Per tal de conèixer el comportament de les proteïnes es poden utilitzar simulacions per computador, ja que és més ràpid i menys costos que no pas fer experiments químics.

2.2 Els aminoàcids

Un aminoàcid és una petita molècula orgànica que conté un grup amino (-NH2) i un grup carboxil (-COOH). Aquests dos grups estan units entre ells per un carboni central (anomenat alfa) i un altre grup que anomenarem cadena lateral (-R). Aquesta cadena lateral diferencia cada aminoàcid de la resta.

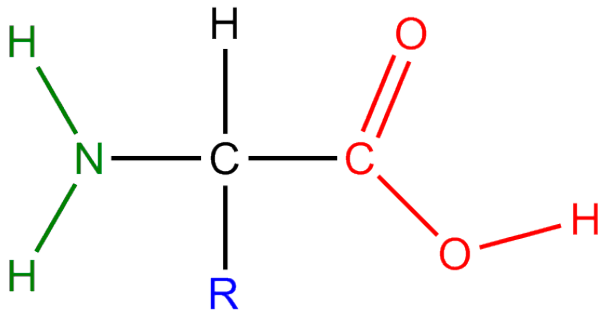


Fig. 1 Aminoàcid esquematitzat.

En els humans trobem fins a 20 aminoàcids diferents, dels quals anomenem essencials a aquells que el cos humà no pot generar per si mateix i ha d'incorporar-los a través de la ingesta d'aliments (per exemple la Isoleucina, Leucina, Lisina, Metionina, Fenilalanina, Treonina, Triptòfan, Valina o Histidina).

Els aminoàcids estan enllaçats entre si mitjançant unions peptídiques (cadena polipeptídica) que es formen entre el grup carboxil del primer aminoàcid el grup amino del segon. En el procés es desprèn una molècula d'aigua (reacció de condensació, veure figura 2).

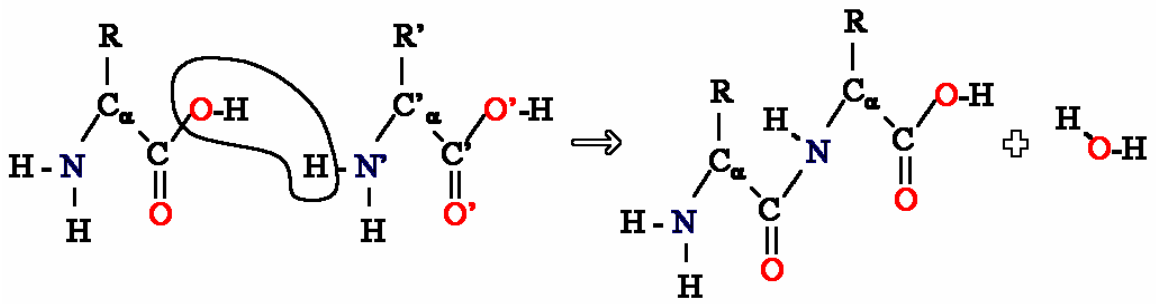


Fig. 2 Formació d'un enllaç peptídic i reacció de condensació.

2.3 Estructura de les proteïnes

Hi ha una relació molt directe entre la forma (l'estructura) de les proteïnes i la seva funció. L'estructura de les proteïnes pot jerarquitzar-se en quatre nivells.

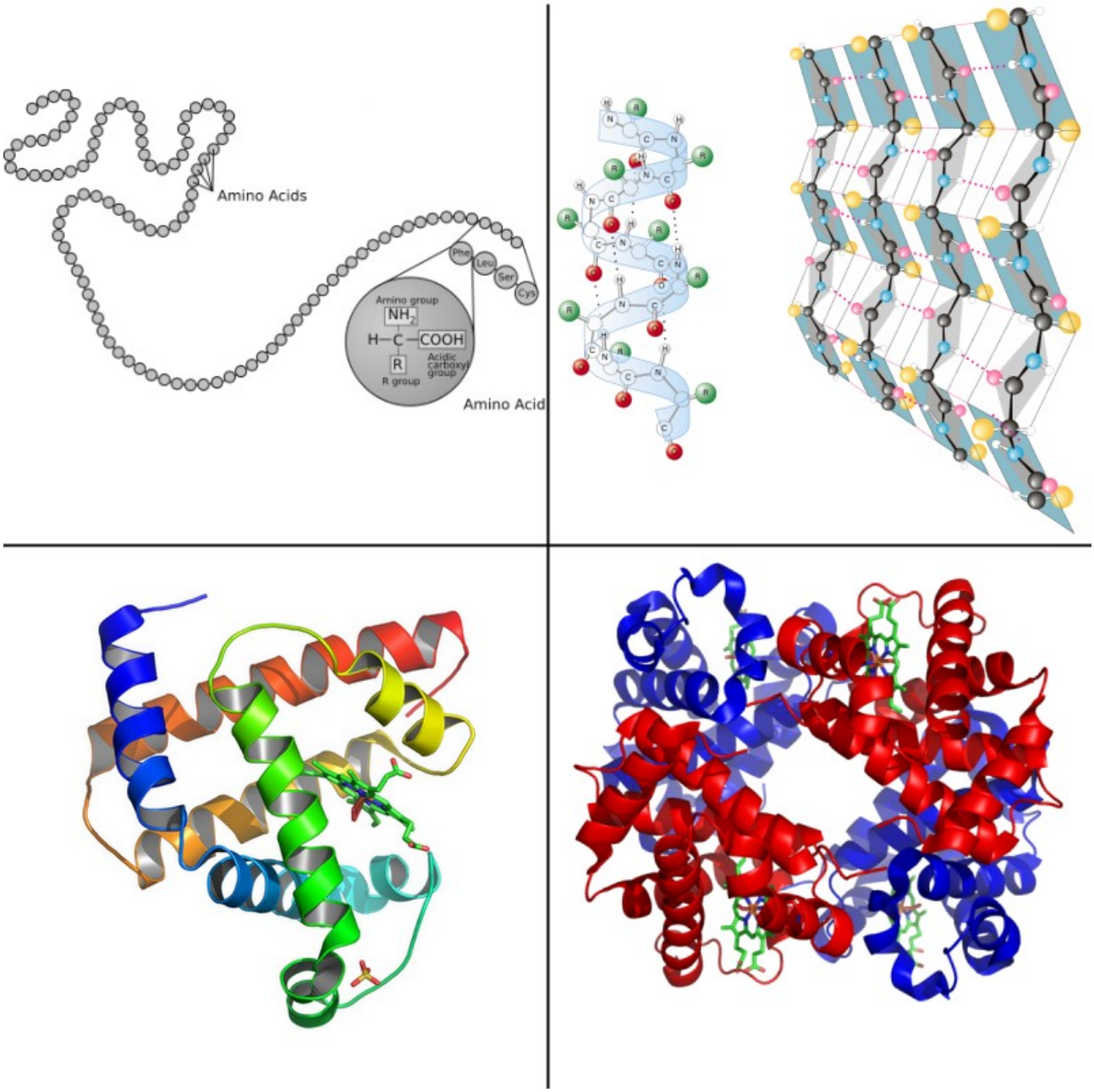


Fig. 3 Estructura primaria (superior esquerra), estructura secundaria (superior dreta), estructura terciària (inferior esquerra), estructura quaternària(inferior dreta)

2.3.1 Estructura primaria

És la seqüencia ordenada d'aminoàcids que formen la proteïna. Ens permet comprendre la seva estructura, el seu mecanisme d'acció i la seva relació amb altres proteïnes amb funció fisiològica similar [1].

2.3.2 Estructura secundaria

És la forma en que es plega la cadena d'aminoàcids de la proteïna. La forma és genera a partir dels ponts d'hidrogen que es formen entre els àtoms dels grups -CO- i -NH- de l'enllaç peptídic. Aquests ponts d'hidrogen donen estabilitat a la estructura secundaria.

Segons el posicionament dels ponts, es generen diferents formes com ara:

- Hèlix alfa
- Fulla beta

2.3.3 Estructura terciària

És l'estructura que defineix la disposició en l'espai de tots els àtoms que composen la proteïna. Li dona la forma tridimensional a la proteïna.

Es distingeixen dos tipus d'estructures terciàries:

- Estructura fibrosa: una de les dimensions és molt més gran que les altres dues.
- Estructura globular: cap dimensió predomina sobre la resta.

L'estructura terciària es genera de forma que els aminoàcids apolars es situen a l'interior i el polars a l'exterior. L'estabilització de l'estructura s'aconsegueix per l'efecte hidrofòbic, les forces de van der Waals, ponts de disulfur (entre aminoàcids de cisteïna convenientment orientats) i mitjançant enllaços iònics.

2.3.4 Estructura quaternària

L'estructura quaternària és la forma que adopten les diferents cadenes polipeptídiques entre si unides mitjançant interaccions no covalents (sense compartir electrons entre àtoms). Per aquesta raó, les proteïnes formades per una sola cadena polipeptídica només arriben a tenir estructura terciària.

2.4 Dinàmica molecular de proteïnes

La dinàmica molecular (DM) és un sistema de simulació per computador dels moviments físics d'àtoms i molècules. Donades les condicions inicials de posició i velocitat, s'utilitzen equacions clàssiques de moviment per tal de determinar/predir la nova posició. Les forces entre les partícules i l'energia estan definits pels camps de força (de l'anglès force fields) de mecànica molecular.

Les simulacions de DM permeten estudiar tant les propietats termodinàmiques del sistema com fenòmens cinètics que succeeixen en una evolució temporal.

En les simulacions, l'interval de temps que transcorre entre mostres és de l'ordre de femto-segons per tal d'observar les noves posicions dels àtoms de les proteïnes. Si volem generar una simulació de magnituds de temps superiors, les simulacions triguen molt ja que es tenen que generar moltes mostres.

Les proteïnes no són objectes estàtics, sinó que estan en continu moviment. Aquestes tècniques per tal d'explorar la relació entre l'estructura, el moviment i la seva funció.

El problema de la DM és la quantitat de recursos computacionals necessaris, ja que les simulacions són sensibles a la mida de la proteïna i són molt lentes de portar a terme [2].

2.4.1 Camps de forces en les proteïnes

Per tal de calcular l'energia potencial de les proteïnes, en la dinàmica molecular s'utilitzen els camps de forces. L'energia potencial depèn de la posició dels àtoms en la proteïna [3].

No hi ha una única fórmula pels camps de força, però acostumen a estar definits per una funció similar a aquesta [4]:

$$V(r) = \sum_{bonds} k_b(b - b_0)^2 + \sum_{angles} k_\theta(\theta - \theta_0)^2 + \sum_{torsions} k_\phi[\cos(n\phi + \delta) + 1]$$
$$+ \sum_{\substack{nonbond \\ pairs}} \left[\frac{q_i q_j}{r_{ij}} + \frac{A_{ij}}{r_{ij}^{12}} - \frac{C_{ij}}{r_{ij}^{6}}\right]$$

Fig. 4 Fórmula energia potencial.

En la equació de la figura 4, es poden veure els termes en que es composa l'energia potencial per aquest camp de forces determinat. Veiem tres sumatoris que representen l'energia dels enllaços covalents (enllaços, angles i torsions) i un degut a les interaccions no covalents.

Els termes d'energia covalents covalents són la suma de :

- l'energia d'enllaç entre àtoms, que va en funció de la distancia entre àtoms i la constant de força k_b definida pels dos àtoms enllaçats consecutius
- l'energia dels angles formats per 3 àtoms consecutius, on θ es l'angle, θ_0 l'angle ideal i k_θ la constant de força definida pels àtoms
- les energies de les torsions del diedre, que representa el moviment de rotació de l'enllaç central (en aquest cas a partir de la funció cosinus)

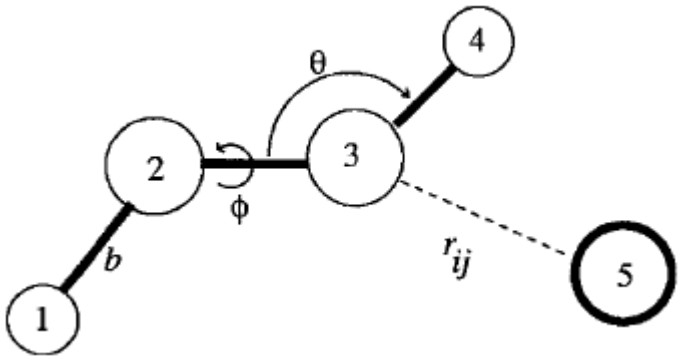


Fig. 5 Energies que intervenen en els camps de forces.

En el càlcul de l'energia de les interaccions no covalents, el primer terme del sumatori $\sum \frac{q_i q_j}{r_{ij}}$ és la força electrostàtica, on q és la càrrega de cada àtom i r la distancia entre ells.

La segona part és la interacció de van der Waals $\sum \frac{A_{ij}}{r^{12}_{ij}} - \frac{C_{ij}}{r^{6}_{ij}}$, que s'origina a partir d'un balanç entre forces repulsives i atractives. Si la distància a la que estan els àtoms és l'equilibri, no hi haurà ninguna força, si estan més allunyats del punt d'equilibri apareixerà una força d'atracció i si estan més pròxims del punt d'equilibri es genera una força de repulsió. A i C són constants definides i r és la distancia entre àtoms

Degut a la gran quantitat d'àtoms, el nombre de possibles parelles d'àtoms és d'ordre quadràtic. Per aquesta raó només tindrem en consideració les parelles d'àtoms que estan dins un radi determinat anomenat distància de tall (en anglès cutoff). Això es pot realitzar perquè l'aportació d'energia dels àtoms que estan allunyats és molt petita i menyspreable.

Existeixen diferents camps de força com OPLS, AMBER o CHARMM; en cada un d'ells s'utilitzen diferents paràmetres i algunes variacions sobre la fórmula d'energia potencial presentada en la figura 4.

3 Software de simulació PELE++

3.1 PELE

PELE (Protein Energy Landscape Exploration) té com objectiu explorar l'espai conformacional de les proteïnes d'una manera computacionalment eficient, gràcies a l'ús del mètode de Monte Carlo[1].

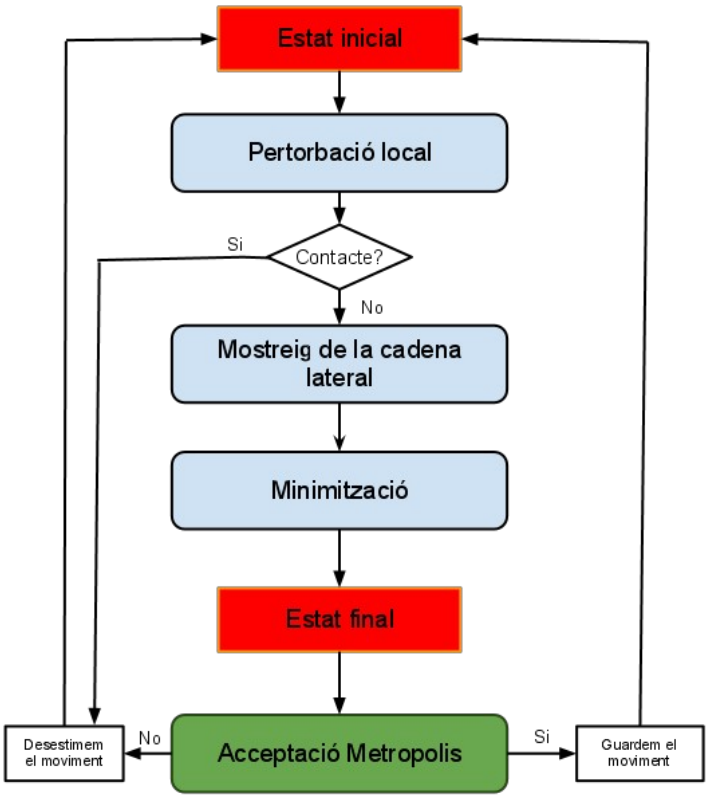


Fig. 6 Diagrama del funcionament de PELE.

Els passos d'una iteració de PELE, donada una conformació inicial de la proteïna, amb una energia potencial inicial associada són:

1 - Pertorbació local
S'aplica una pertorbació (rotació i translació) al lligand[2] de la proteïna i/o a la proteïna (actualment mitjançant l'aplicació de modes normals). Si un cop finalitzat el procés de pertorbació hi ha contacte entre el lligand i el backbone[3] de la proteïna, la pertorbació es rebutja.

2 - Mostreig de la cadena lateral
La disposició de les cadenes laterals afecta molt a l'energia final de la proteïna. En aquesta etapa es modifica la posició de les cadenes laterals per poder disminuir l'energia.

1 El mètode de Monte Carlo consisteix en definir un domini d'entrades i seleccionar-ne una a l'atzar a partir d'una funció de probabilitat.
2 El lligand es una substància que s'adhereix a una biomolècula, per exemple l'oxigen, els lípids ...
3 El backbone és l'esquelet central de la proteïna.

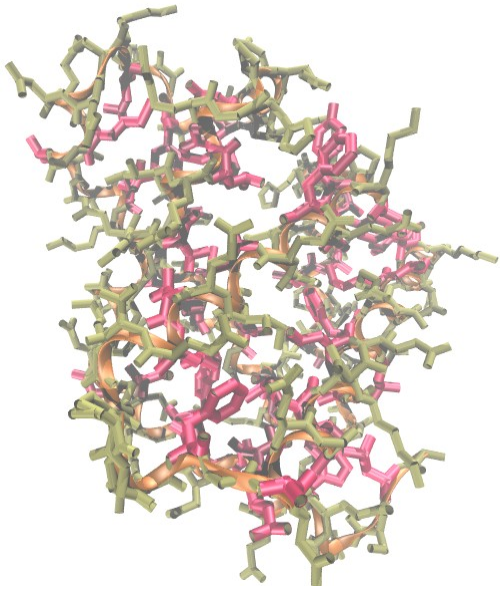


Fig. 7 Myoglobina on es poden apreciar les cadenes laterals (en verd les hidrofíliques i vermell les hidrofóbiques).

3 - Minimització
S'aplica l'algorisme de minimització Newton Truncat, que modifica l'estructura per poder abaixar encara més l'energia de la conformació.

4 - Acceptació de Metròpolis
S'aplica el criteri de Metròpolis sobre la nova conformació de la proteïna:
$\Delta V < 0$

$$\exp\left(\frac{-\Delta V}{K_b T}\right) < R$$

on K_b és la constant de Boltzmann, R un número aleatori, T la temperatura elegida en la simulació i ΔV es la diferencia d'energia entre la conformació inicial i la final.

Quan es compleixen les dues inequacions anteriors (la reducció de l'energia en la nova conformació és suficient), s'accepta la nova conformació i la aquesta serà usada com a conformació inicial en una nova iteració de l'algorisme. En cas contrari es desestima la pertorbació aplicada.

Com a mètode estocàstic[4], les captures no són generades per una equació de moviment de Newton determinista, sinó que es un cerca aleatòria en l'escenari preestablert. Per aquesta raó l'exploració és més ràpida que en DM.

3.2 PELE++

PELE++ es una reescriptura en el llenguatge de programació C++ del software PELE, inicialment escrit en el llenguatge de programació FORTRAN. En aquesta nova versió s'ha tingut molta cura en aplicar programació orientada a objectes, patrons de disseny, bona cobertura de tests, i, en definitiva, usar les eines que l'enginyeria del software d'avui dia ens aporta. L'objectiu final és facilitar les fases de manteniment i l'actualització del codi, així com millorar la seva eficiència mitjançant l'ús de nous algoritmes experimentals i la paral·lelització del software.

4 Comportament no determinista

3.2.1 Profiling

S'han extret profiles del programa utilitzant tres proteïnes minimitzades de 1.231 (1) , 14.132 (2) i 31.313 (3) àtoms. Els profiles s'han obtingut amb el programa *gprof* . La informació més rellevant és la següent:

	Proteïnes		
	1	2	3
MultiScaleEnergyAlgorithm::calcOnlyLongNonBondingEnergyAndGradient	25,48	30,83	27,75
MultiScaleEnergyAlgorithm::calcNonBondingEnergy	5,46	3,73	3,35
MultiScaleEnergyAlgorithm::calcOnlyShortNonBondingEnergyAndGradient	0,92	1,17	0,00
EnergyAlgorithm::calcCovalentEnergyGradient	3,21	4,89	15,88
Solvent::updateAlphasSasaAndEnergies	54,08	48,20	40,26
Total	89,15	88,82	87,24
Total NonBonding	31,86	35,73	31,10
Ratio Solvent:NonBonding	1,70	1,35	1,29

Taula 1 Percentatges de temps emprat en el càlcul de l'energia i el solvent.

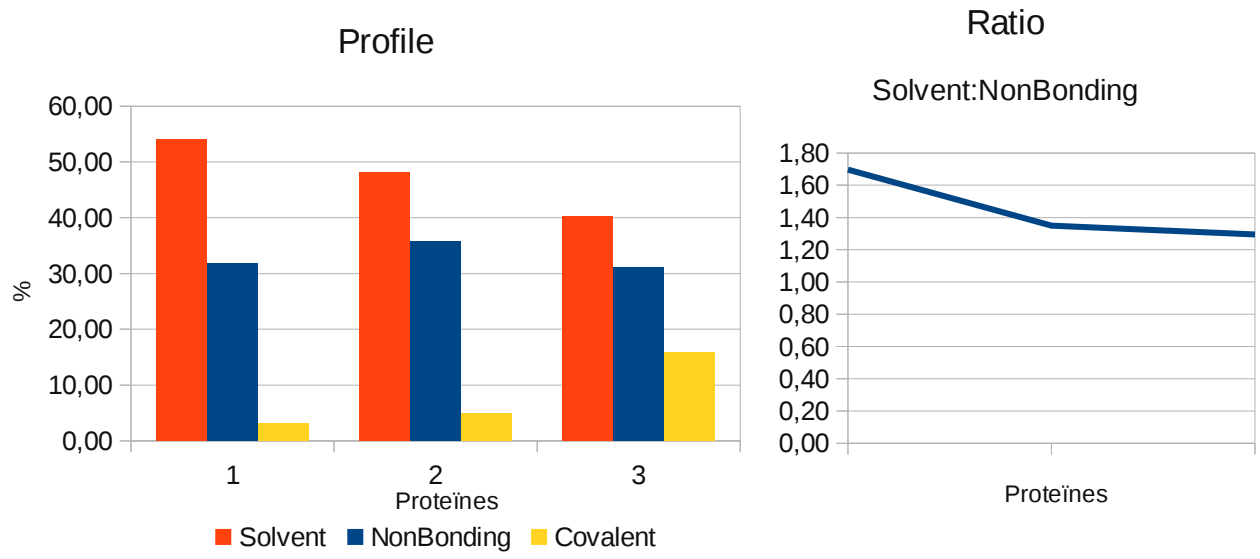


Fig. 8 Gràfics representant el temps emprat en PELE++ en els diferents càlculs, i el ràtio entre el càlcul del solvent i l'energia de les interaccions no covalents.

La major part del temps de l'execució de PELE++ es reparteix entre el càlcul de l'energia de les interaccions no covalents i el solvent. La funció que calcula l'energia de les interaccions no covalents utilitza els camps de forces de dinàmica molecular per obtenir l'energia.

A mesura que la proteïna creix, s'observa com el temps emprat en el solvent disminueix, del 54% al 40%, mentre el càlcul de les energies no covalents que tendia a créixer en les dues primeres proteïnes, decreix en la més gran. Això es degut al fort creixement del percentatge de temps d'execució degut al càlcul de l'energia covalent en aquesta última proteïna, que fa decréixer el pes dels altres dos càlculs. Tot i així, el ràtio entre el temps en el càlcul del solvent i el de l'energia no covalent mostra com el solvent va perdent importància quan augmentem la mida de les proteïnes.

El principal coll d'ampolla és el càlcul del solvent, però degut a la seva complexitat s'ha desestimat realitzar els càlculs a la GPU. Com que els càlculs d'energia de les interaccions no covalents (INC) representen el segon grup de funcions que més temps s'executen, i al fer càlculs més senzills, seran

les millors candidates per la paral·lelització.

El fet de que el càlcul de les energies de les interaccions no covalents sigui un dels colls d'ampolla és un problema recorrent en dinàmica molecular, on també s'utilitzen aquest tipus de càlculs [5].

3.2.2 Aïllament de les funcions objectiu

En el profile, com hem vist abans, destaquen tres funcions relacionades amb l'anàlisi de les interaccions no covalents: *calcNonBondingEnergy* , *calcOnlyLongNonBondingEnergyAndGradient* i *calcOnlyShortNonBondingEnergyAndGradient* . La primera funció només realitza el càlcul de l'energia de les INC, mentre que les altres dues funcions calculen energia i gradient alhora. Aquestes tres funcions són implementacions de dues funcionalitats, la especificació de les quals seria:

```
Funció calcEnergy ( e real[] coords, e NonBondPair[] nbi) retorna real energia
//Pre: coord i nbi estan inicialitzats
//Post: retorna l'energia acumulada de totes les intereccions contingudes en nbi
```

Codi 1 Especificació de calcEnergy.

```
Funció calcEnergyGradient ( e real[] coords, e NonBondPair[] nbi, s real[] grad) retorna
real energia
//Pre: coord i nbi estan inicialitzats
//Post: retorna l'energia acumulada de totes les intereccions contingudes en nbi i el
gradient en el vector grad
```

Codi 2 Especificació calcEnergyGradient.

Els atributs que trobem en les funcions són els següents:

- *coords* : és un vector de reals en el qual trobem, organitzades en tripletes, les coordenades de cada àtom. Per exemple, si tenim dos àtoms en el sistema, el nostre vector de coordenades tindrá sis posicions, on les tres primeres correspondran a les coordenades x , y , z del primer àtom i les tres següents a les coordenades x , y , z del segon.

Àtom 1			Àtom 2			...	Àtom n		
x	y	z	x	y	z	...	x	y	z

Fig. 9 Estructura del vector coords.

- *nbi* : és un vector d'objectes de tipus *NonBondPair* . Cada un d'aquests objectes representa una interacció no covalent. Està formada per un parell de punters a estructures de tipus *Atom* , i les constants electrostàtiques i de van der Waals que defineixen la interacció.

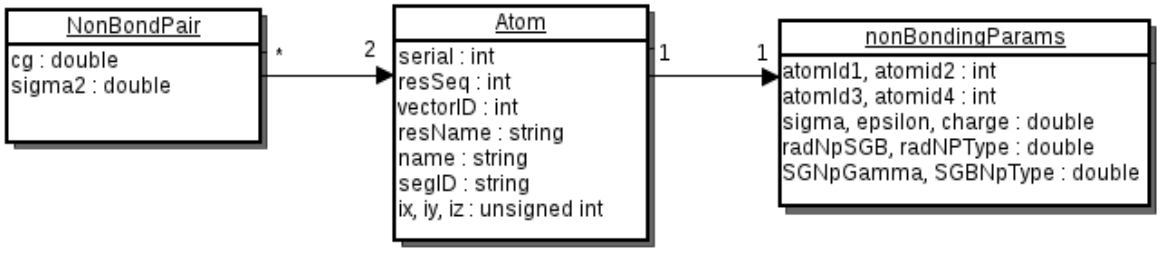


Fig. 10 Diagrama de classes dels objectes NonBondPair, Atom i nonBondingParams.

- *grad* : és un vector de reals on cada àtom acumula el seu gradient per a cada coordenada i es va completant a mesura que es van processant les seves parelles. El vector segueix la mateixa organització que el vector *coords* , però enlloc de contenir la posició de cada coordenada, conté el gradient de cada coordenada.

S'han aïllat aquestes dues funcions i s'ha creat un entorn de test on desenvolupar i provar les diferents tècniques proposades en el projecte. En aquest test s'introdueixen les interaccions no covalents de la molècula (en forma de llista de *NonBondPair*) de la qual volem calcular l'energia i el gradient, i el resultat que ens hauria de donar si s'han modificat correctament les funcions anteriors (tot el subsistema de test queda explicat en detall a l'apartat 5.1.1). Així no haurem de modificar PELE++ directament, nomes les funcions objectiu de les quals prendrem les mesures de temps per veure com varia el rendiment. L'aïllament de les funcions ens permetrà treballar de forma més ràpida i eficient, ja que ens limiten les dependències amb la resta del codi de PELE++.

3.2.3 Generació de les interaccions no covalents a PELE++

En PELE++ hi ha implementades dues estratègies per obtenir les llistes de les INC: per força bruta i multi-escala. L'ús d'una estratègia o altre fa variar el temps necessari per generar les llistes i l'espai necessari en memòria (ja que variarà el nombre de parelles generades).

Força bruta
Considerem totes les possibles parelles d'àtoms, i si la distancia entre aquests està dintre d'una distància de tall preestablerta, aquesta interacció entrarà dins el càlcul.

Multi-escala
Es un mètode més complex que es basa en les propietats físiques dels àtoms. Per seleccionar les interaccions s'usen fins a sis distàncies de tall, segons la distància de la interacció (curta o llarga) i la càrrega dels àtoms (positiu, negatiu, neutre).

3.2.4 Les interaccions no covalents en l'etapa de minimització

La generació de les parelles i els càlculs de l'energia es duen a terme en l'etapa de minimització. Abans de cridar a la funció del càlcul de l'energia i el gradient s'han de crear les llistes d'INC.

Per a cada iteració de PELE es realitzen tres minimitzacions. La minimització és un procés molt complex on la funció *calcEnergyGradient* s'usa repetidament. En cadascuna de les minimitzacions es segueix l'algorisme del codi 3.

```
...
mentre i < N fer
      ...
      generar parelles();
      actualitzar coordenades();
      calcEnergyGradient();
      ...
      mentre j < M fer
            ...
            actualitzar coordenades();
            calcEnergyGradient();
            ...
            j++;

      fmentre

      i++;
fmentre
...
```

Codi 3 Fragment en pseudocodi del bucle de la funció de minimització.

Els valors mínims i màxim són 1 i 65 per N, i entre 20 i 50 per M.

4 Paral·lelització

En el passat els ordinadors només disposaven d'un processador i l'execució del codi era seqüencial. Avui en dia disposem de sistemes multiprocessador i podem executar en un mateix instant més d'una instrucció. Per aquesta raó hem de crear codi que aprofiti el fet de tenir més d'un processador, paral·lelitzant.

Hi ha diferents mètodes a l'hora de paral·lelitzar segons l'arquitectura sobre la qual treballem. Aquestes arquitectures són molt variades i poden anar des de d'un conjunt de processadors d'ús general (CPUs) a processadors d'ús específic com els processadors de les targetes gràfiques (com les GPUs[5]) o sistemes heterogenis que combinen diferents arquitectures, per exemple CPUs + GPUs.

En aquest projecte farem la paral·lelització del codi de PELE++ en GPU, i ens centrarem en dos frameworks per a la programació de GPUs, CUDA i OpenCL.

4.1 Arquitectura GPU

4.1.1 Inicis de les GPUs

Fins l'any 2006, les GPUs acostumaven a tenir les funcions fixades, ja que s'utilitzaven només per generar gràfics 3D. Les etapes del pipeline típic per a les quals estaven dissenyades les GPUs era el següent:

- Operacions de vèrtexs: es computen els vèrtexs dels elements a mostrar en l'escena.
- Assemblatge de primitives: els vèrtexs s'uneixen formant triangles que representaran l'objecte 3D.
- Rasterització: es generen els fragments, que són els píxels en pantalla que ocuparan els triangles.
- Operacions de fragments: es pinten els fragments.
- Composició: es munten els fragments en l'espai, es creen les oclusions ...

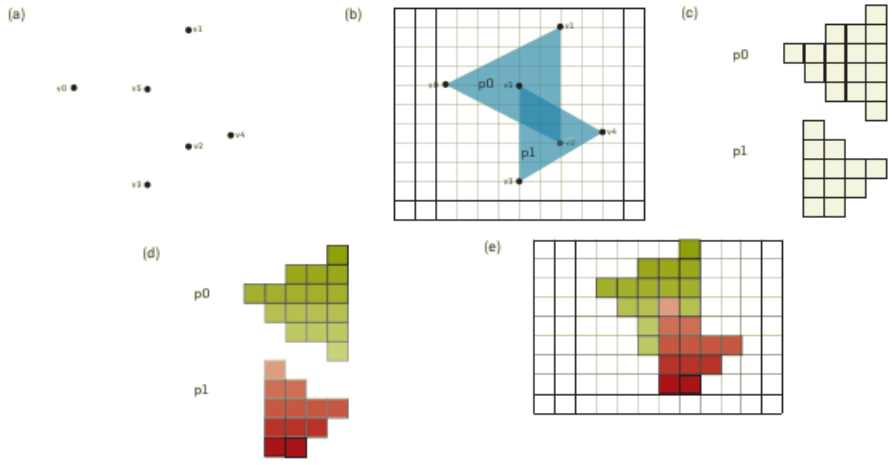


Fig. 11 a) Operacions de vèrtexs, b) Assemblatge de primitives, c) Rasterització, d) Operacions de fragments, e) Composició [13]

5 La tècnica d'usar la GPU amb finalitats “no gràfiques” s'anomena GPGPU.

Les etapes del pipeline eren configurables però no programables.

4.1.2 Arquitectura actual: Shaders unificats

A dia d'avui, el pipeline ha canviat, i enlloc de tenir unitats hardware fixades amb funcions específiques per vèrtexs i fragments, tenim només un tipus d'unitat hardware programable. És el programador el que acaba determinant quantes unitats de càlcul vol assignar a operacions de vèrtexs o fragments.

Aquesta nova estructura fa que la GPU divideixi els recursos del processador en diferents etapes, i el pipeline en espai i no temps. Per exemple, utilitzant una targeta gràfica amb les unitats hardware fixades, 24 unitats per operacions de vèrtexs i 48 unitats per fragments, no podíem variar l'assignació d'aquests recursos que s'executaven en cascada, en canvi, amb una targeta gràfica moderna, podríem tenir 72 unitats de càlcul que repartiríem de la forma més adequada a les nostres necessitats, i s'executarien de forma paral·lela.

Com es permeten diferents camins d'execució per a cada unitat de càlcul, es necessita una gran part de hardware de control i per aquesta raó els fabricants agrupen les unitats de càlcul en blocs que s'executen en paral·lel.

Les GPUs tenen un througput molt elevat ja que poden fer moltes operacions simultàniament. També tenen un gran ample de banda amb la memòria que subministra les dades, però la latència d'accés és elevada.

Per tal de pal·liar l'elevada latència utilitzen el multithreading com a mecanisme principal per disminuir el temps d'espera d'un thread quan ha d'accedir a memòria. Utilitzant aquest sistema, un altre thread aprofita la unitat de càlcul mentre l'anterior esta bloquejat accedint a memòria i per tant no la pot utilitzar.

La GPU manté funcions hardware específiques fixades necessàries en la generació de gràfics ja que no hem d'oblidar quina és la seva concepció inicial.

En la taula 2, es pot observar les diferents arquitectures ja siguin CPUs o GPUs, amb la seva distribució de les unitats de càlcul i el ràtio dels threads-hardware entre tots els que pot executar (1 significa que no té multithreading):

Tipus	Processador	Nuclis	ALUs/Nucli	Max T
GPU	AMD Radeon HD 4870	10	80	25
	Nvidia Geforce GTX 280	30	8	128
CPU	Intel Core 2 Quad	4	8	1
	STI Cell BE	8	4	1
	Sun Ultra SPARC T2	8	1	4

Taula 2 Taula de característiques de diferents arquitectures.

En resum, una GPU es pot veure com un conjunt de multiprocessadors, on cada multiprocessador esta format per processadors escalars. Formen una arquitectura SIMT (Single Instruction Multiple Thread) ja que tots els processadors de dins el multiprocessador executen les mateixes instruccions.

L'arquitectura interna de les GPUs varia segons el fabricant, però segueixen una estructura similar.

4.1.3 GPGPU

Gràcies a l'arquitectura de les GPUs ens trobem uns processadors amb moltes unitats de càlcul en

paral·lel que podem utilitzar per altres funcions que no siguin gràfiques.

El nou model de programació és el SPMD (Single Program Multiple Data). Es similar al les instruccions vectorials o SIMD (Single Instructions Multiple Data) que incorporen moltes CPUs, com les instruccions SSE de Intel i AMD, on amb una instrucció s'executa la mateixa operació en un conjunt de dades. En SPMD enlloc de ser una instrucció és un programa que s'executa sobre un conjunt de dades.

La programació s'estructura de la següent forma:

- El programador defineix el domini estructurat en un conjunt de threads.
- Crea un programa que s'executarà en tots els threads (SPMD).
- Cada thread executarà el programa guardant el resultat generat en un buffer.

Les tècniques habituals a tenir presents en sistemes d'arquitectures paral·leles són les següents:

- Scatter/Gather: “escriure a” o “llegir des de” les diferents seccions de memòria amb les que treballarem. Ja hem anunciat anteriorment que tindrem més d'un espai de memòria amb diferents característiques cada un d'ells.
- Map: consisteix en trossejar la feina i enviar-la a computar en diferents unitats.
- Reduce: recollir la feina realitzada per cada unitat en l'etapa map i reduir-la a un element.
- Scan: partint d'un vector A, retornarem un vector B on cada B[i] serà una reducció de A[1..i].

4.2 CUDA

El llenguatge CUDA és una extensió del llenguatge C que incorpora noves instruccions per tal de treballar amb la GPU. Va ser creat per Nvidia i els programes en CUDA només poden ser executats en el seu hardware.

En CUDA tenim dues APIs:

- una API de baix nivell anomenada Driver
- una API de nivell superior anomenada Runtime

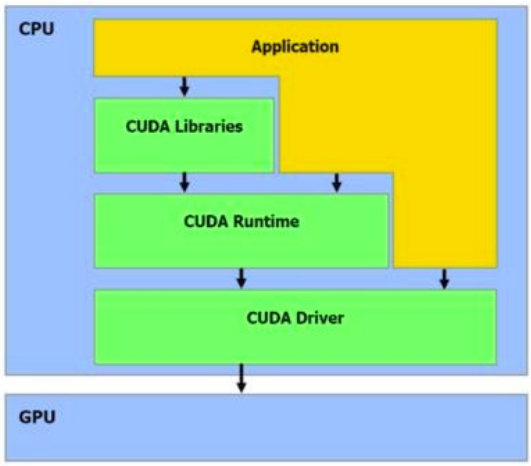


Fig. 12 Capes en l'estructura CUDA [14].

Les diferències entre les dues API segons la guia de programació de CUDA són que el Runtime, el qual utilitzarem en aquest projecte, és més senzill i d'utilitzar ja que les inicialitzacions de l'entorn

CUDA ja estan fetes, per tant només tenim que cridar i implementar les funcions que volem executar en la GPU. El Driver en canvi permet més control i configuració, però necessita molt més codi i és més difícil de programar i debugar.

En aquest projecte utilitzarem el Runtime.

4.2.1 Estructura

4.2.1.1 Hardware

Abans de començar a programar, hem de tenir present el hardware del sistema. La figura 13 és un esquema d'una targeta gràfica Tesla C1060 de Nvidia, on podem veure com la CPU esta connectada a la targeta gràfica, que conté la GPU i la memòria. A l'interior de la GPU hi trobem molts multiprocessadors, on cada un d'ells té una petita memòria que comparteixen els 8 processadors que hi han al seu interior. Al mateix temps esta connectada a la memòria (Global Memory) de la targeta gràfica. Conèixer el nombre de processadors per bloc, la mida de les memòries i altres característiques que pugui tenir la targeta gràfica on volem executar el nostre codi, ens permetrà optimitzar millor el nostre codi per tal de treure el màxim rendiment possible.

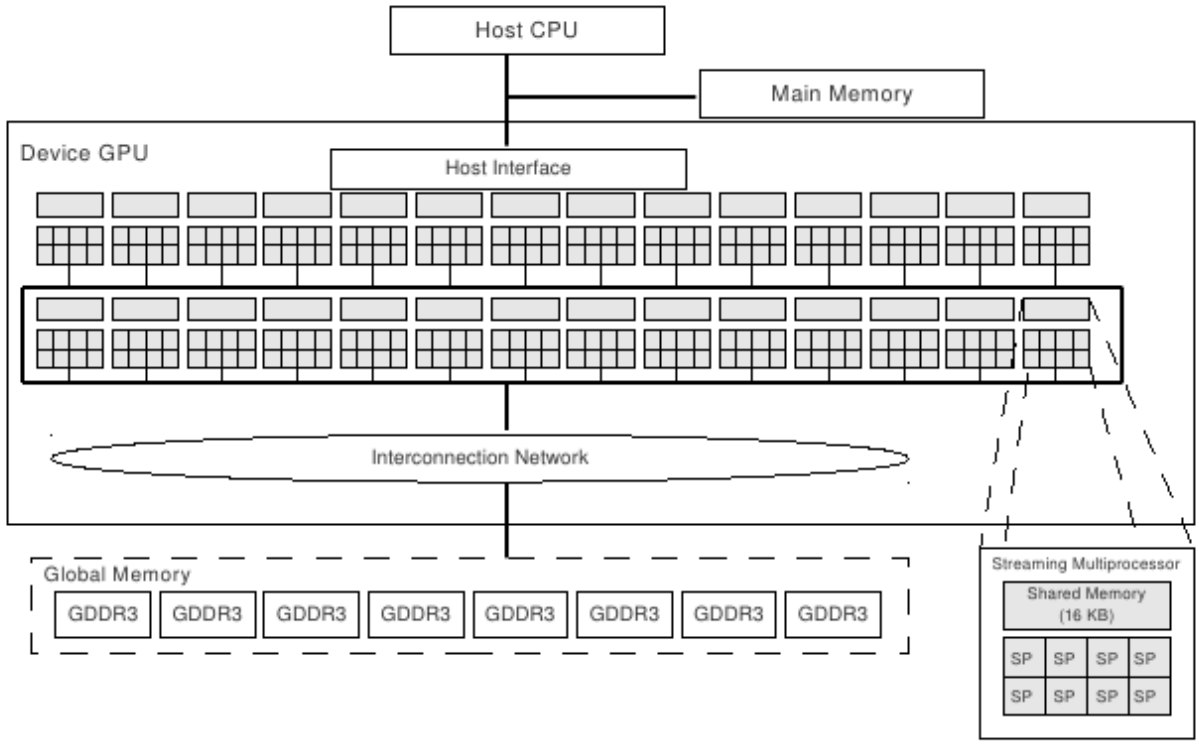


Fig. 13 Arquitectura de la GPU connectada a la CPU i memòria [15].

4.2.1.2 Programa CUDA

Un programa en CUDA té dues parts, el codi que s'executa en la CPU (host) i el codi que s'executa en la GPU (device), anomenat kernel.

En el kernel tenim dos nivells de paral·lelisme, a nivell de bloc i de thread.

Un block equival a un multiprocessador el qual conté threads que s'executen dins seu. El “warp size” és el nombre màxim de threads que pot gestionar cada multiprocessador (varia segons la versió de CUDA). Cada multiprocessador crea, gestiona, executa i planifica els seus threads en els seus processadors. Els threads entre multiprocessadors no es poden comunicar.

Els blocs d'un kernel estan agrupats en una graella (grid). Des de la versió 2.x de CUDA es possible executar més d'un kernel concurrentment.

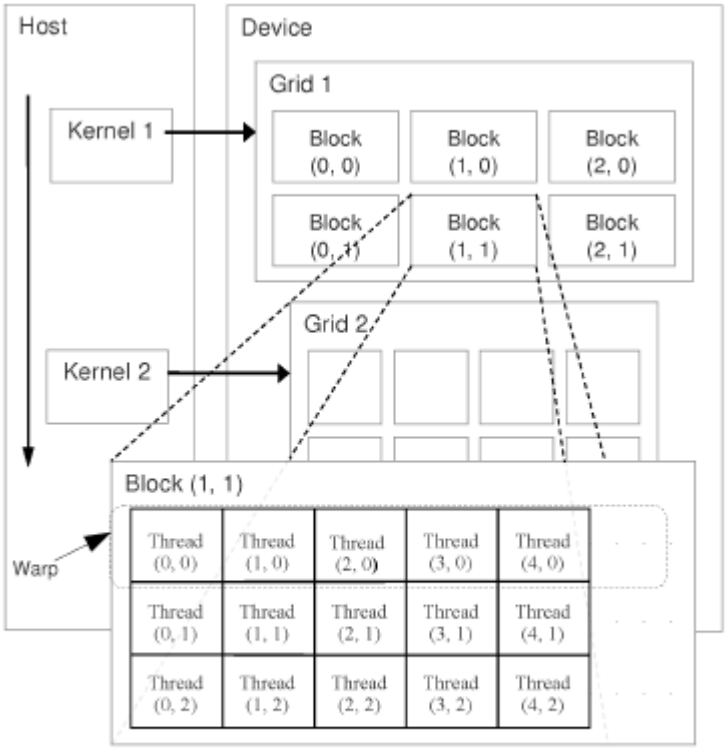


Fig. 14 Model de programació CUDA [15].

El programador declara en el codi del host quants blocs i threads li assigna al kernel.

En el següent exemple de codi en CUDA realitzem la suma de dos vectors, assignant a cada bloc una posició del vector.

```
01 // Kernel = codi que s'executarà en els processadors de la gpu
02 __global__void add( int *a, int *b, int *c){
03    int tid = blockIdx.x;
04    if(tid < N)
05          c[tid]=a[tid] + b[tid]
06 }
07
08 // Main = codi que executa la cpu
09 int main{
10    int a[N], b[N], c[N];
11    int *dev_a, *dev_b, *dev_c;
12    //Reservem memoria a la gpu pels vectors a, b i c
13    cudaMalloc( (void**)&dev_a, N * sizeof(int)));
14    cudaMalloc( (void**)&dev_b, N * sizeof(int)));
15    cudaMalloc( (void**)&dev_c, N * sizeof(int)));
16    generarAiB(a, b);
17    //Copiem els vectors a i b a la memoria de la gpu
18    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
19    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
20    //Executem el kernel
21    add<<<N,1>>>(dev_a, dev_b, dev_c);
22    //Recollim el resultat i el desem al vector c
23    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
24    //Lliberem la memoria de la gpu
25    cudaFree (dev_a);
26    cudaFree (dev_b);
27    cudaFree (dev_c);
28 }
```

Codi 4 Exemple de codi en CUDA.

El codi anterior es un bon exemple dels passos a seguir al fer un programa que usi CUDA:

1. Reservar espai a la memòria de la targeta gràfica (línies 13-15)
2. Copiar les dades a la memòria de la targeta gràfica (18-19)
3. Executar el kernel (21)
4. Recollir el resultat copiant les dades de la memòria de la targeta gràfica a la CPU (23)
5. Alliberar la memòria de la targeta gràfica (25-27)

4.2.2 Model de memòria

La GPU té diferents memòries a les quals podem accedir en els kernels:

- Memòria global: memòria principal de la targeta gràfica.
- Memòria local del thread: nomes pot accedir el thread.
- Memòria compartida pels threads: memòria que comparteixen els threads d'un bloc.
- Memòria constant: nomes lectura, accessible pels threads de forma simultània per a la meitat d'un warp i després per l'altre meitat.
- Memòria de textures: només de lectura i esta optimitzada per aprofitar la localitat espacial en 2D.

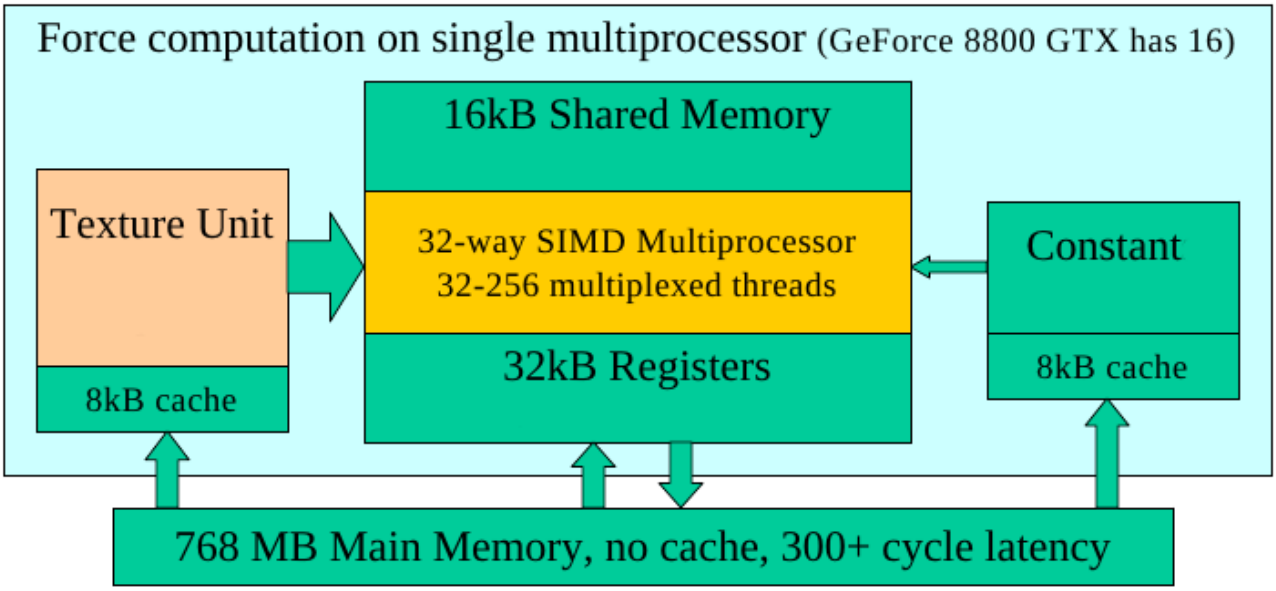


Fig. 15 Memòria d'una targeta GeForce 8800 GTX [16].

4.2.3 Sincronisme

4.2.3.1 A nivell de threads

Els threads dins un block es poden sincronitzar entre ells utilitzant la funció *syncthreads* . Quan un thread arriba a la funció *syncthreads* s'atura, i fins que l'últim thread del block no arriba a la funció no es continua amb l'execució de tots els threads.

4.2.3.2 A nivell de funcions

En CUDA, hi ha funcions síncrones (fins que no acaba l'execució de la funció, la CPU no es continua amb la següent instrucció) i asíncrones (es continua amb la següent instrucció sense esperar a que acabi la funció asíncrona). D'aquesta forma es pot aconseguir que la CPU i la GPU treballin simultàniament. Hi ha funcions com el *cudaMemcpy* i el *cudaMemcpyAsync* que tenen dues versions, en canvi d'altres com el llançament dels kernels són sempre asíncrons.

Les funcions dins la GPU s'executen de forma ordenada, i si la CPU fa una crida a una funció CUDA síncrona i hi ha funcions asíncrones executant-se, fins que no acabin totes la CPU no passarà a la següent instrucció.

Es poden crear punts de sincronisme amb la funció *cudaThreadSynchronise* o be utilitzant esdeveniments amb *cudaEventRecord* i *cudaEventSynchronise* .

4.2.4 Versions

Existeixen diferents version de CUDA segons la capacitat de càlcul de la targeta gràfica. No es pot executar una aplicació CUDA compilada en una versió superior a la que admet la targeta gràfica.

4.3 OpenCL

OpenCL és un framework de programació per a sistemes híbrids per tal d'explotar la paral·lelització.

Va ser inicialment creat per Apple i després el va proposar al grup Khronos per tal que l'estandarditzessin i així crear un estàndard obert i lliure.

Les diferencies més rellevants si el comparem amb CUDA són:

- No es propietari i pot ser executat en qualsevol hardware que el suporti.
- Esta dissenyat per sistemes híbrids, no només per GPUs com CUDA.
- OpenCL no té accés a la memòria de textures de la GPU a diferencia de CUDA.

En OpenCL tot dispositiu hardware que pugui ser utilitzat per realitzar càlculs i suporti OpenCL serà tractat com un device, ja sigui una CPU, GPU, DSP o qualsevol altre tipus de processador.

Es pot fer un mapeig quasi directe entre els noms en CUDA i OpenCL. Per tal de no crear confusió, en el llarg del projecte s'utilitzarà el terme CUDA.

Terme CUDA	**Terme OpenCL**
GPU	Device
Multiprocessor	Compute Unit
Scalar core	Processing element
Global memory	Global memory
Shared (per-block) memory	Local memory
Local memory	Private memory
Kernel	Program
Block	Work group
Thread	Work item

Taula 3 Equivalència de nomenclatura entre CUDA i OpenCL.

La memòria dels dispositius on s'executen els kernels, en el nostre cas la memòria de la targeta gràfica, en OpenCL s'anomena buffer.

L'estructura d'un programa en OpenCL és la següent:

1. Creació del context: Es selecciona la plataforma de treball desitjada entre les disponibles en el sistema i es configurà el context.
2. Creació de la cua de comandes/execució: Es selecciona el dispositiu del context desitjat i es crea una cua d'execució per a aquest dispositiu. Totes les operacions (copiar dades a memòria, executar kernel ...) que volem realitzar amb aquest dispositiu han d'utilitzar aquesta cua.
3. Creació del programa: Es llegeix el fitxer que conte els kernels.
4. Creació del kernel: S'indica la funció (el kernel) a executar del programa.
5. Crear els buffers a memòria: Reservar espai i guardar a la memòria les dades necessàries pel kernel.
6. Indicar els buffers que utilitzarà el kernel.
7. Posar el kernel a execució encuant-l'ho a la cua de comandes.
8. Recollir els resultats dels buffers.
9. Alliberar recursos.

Com s'observa es necessita molt més codi per realitzar la inicialització (punt del 1 al 4) del sistema que no pas utilitzant el Runtime de CUDA on no hem de fer res. OpenCL s'assembla més al Driver de CUDA.

La idea de treball i el model de programació amb OpenCL és molt similar a CUDA. Les funcions CUDA que utilitzarem tenen el seu equivalent en OpenCL.

5 Entorn de desenvolupament

El desenvolupament i execució es realitzarà en el clúster del BSC anomenat Minotauro, al qual tindrem accés utilitzant ssh.

5.1 Software

En la sessió d'usuari de Minotauro s'ha instal·lat l'entorn de programació Eclipse amb els plugins de C++ (CDT) i CUDA (http://ydl.net/eclipse_cuda_plugin/helios/)

Els compiladors de C, C++ i CUDA ja estaven instal·lats.

Sistema operatiu:

Red Hat Linux (Kernel 2.6.32-71.el6.x86_64 SMP)

Compiladors:

C i C++: gcc 4.4.4
CUDA: nvcc 4.0

IDE:

Eclipse Helios 3.6.2 amb CDT 7.0.2

En el compilador s'utilitza el flag -O3 d'optimització.

El driver d'OpenCL instalat és el de NVIDIA.

5.1.1 Test

Tal i com s'ha introduït en l'apartat 3.4, treballarem en un sandbox (entorn de proves). En aquest entorn es defineix un test on es realitza de PELE el càlcul de l'energia i el gradient de les INC mitjançant les funcions *calcEnergy* i *calcEnergyGradient* .

Dins el test crearem una classe nova per a cada implementació de les funcions d'energia. Tindrem tres implementacions, la versió seqüencial que s'executa a la CPU, i les noves versions CUDA i OpenCL. Creant una herencia de *EnergyCalculator* podrem compartir atributs comuns entre versions.

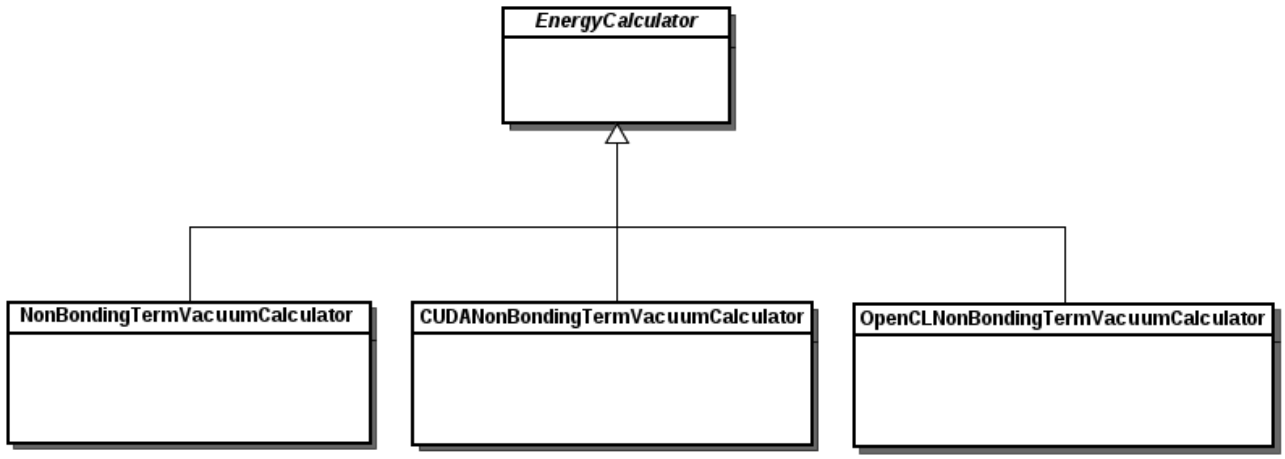


Fig. 16 Noves classes que crearem, CUDANonBondingTermVacuumCalculator i OpenCLNonBondingTermVacuumCalculator.

Les funcions que realitza el test són:

- Cridar les funcions *calcEnergy* i *calcEnergyGradient* .
- Comparar els resultats de les dues funcions per verificar-ne el seu correcte funcionament i, en cas d'error, mostrar quines dades no són com s'esperaven. El resultat de l'energia i gradient que han de donar s'han extret de la versió estable de PELE++.
- Mesurar el temps transcorregut.

Per tal de realitzar les seves funcions i poder-les desacoblar de la resta del codi de PELE, al test se li introdueixen els àtoms, les INC no covalents de la proteïna i els resultats de l'energia i el gradient. Per a cada joc de proves, aquests quatre fitxers han d'estar dins un directori anomenat DataX (X es el número del joc de proves) en el mateix directori on esta l'executable del test.

També li hem d'indicar quina implementació volem que executi: seqüencial, CUDA o OpenCL. El disseny ens permet canviar la implementació en temps d'execució.

Per executar el test introduirem:

```
>./test_pele <implementació> <número del joc de proves>
```

Resultat execució:

```
[bsc72773@nvb127 Release]$ ./test_pele c 13
CUDA CALCULATOR
Number of threads per block 32
Reading coordinates...
Reading atoms...
Reading pairs...
CopyCoords took 0.001 seconds (0.001s mean exec. time)
AtomVector took 0.001 seconds (0.001s mean exec. time)
NBIVector took 0.092 seconds (0.092s mean exec. time)
MapVectors took 0.016 seconds (0.016s mean exec. time)
EnergyCalculation took 0.004 seconds (0.004s mean exec. time)
Energy (calcEnergy): 68914.1
EnergyGradientCalculation took 0.019 seconds (0.019s mean exec. time)
Energy (calcEnergyGradient): 68914.1
- Energy OK
- Energy OK
- Gradient OK
OK
Whole_Test took 2.469 seconds (2.469s mean exec. Time)
```

5.2 Hardware

Minotauro és un supercomputador format per 128 blades Bull B505. Cada blade esta format per dos processadors Intel Xeon E5649 de 64 bits, dues NVIDIA Tesla M2090, 24GB de memòria ram i un disc dur SSD de 250GB. La capacitat total de càlcul es de 103,2 teraflops amb puntes de fins a 185,78 teraflops. És el supercomputador més eficient d'Europa [6].

5.2.1 Informació CPU

Dades més rellevants de la cpu (/proc/cpuinfo):

```
model name   : Intel(R) Xeon(R) CPU E5649  @ 2.53GHz
cpu MHz      : 2533.101
cache size   : 12288 KB
cpu cores    : 6
bogomips     : 5066.54
```

5.2.2 Informació GPU

Les característiques més rellevants de la targeta gràfica Tesla M2090 a la pàgina web de NVIDIA són les següents:

- 665 GFlops màxim en doble precisió
- 6 GB memòria (ECC off)
- 177 GB/sec ample de banda de la memòria (ECC off)
- 512 CUDA cores (16 multiprocessadors amb 32 processadors CUDA cada un)
- Versió CUDA suportada 2.0

Utilitzant la funció *cudaGetDeviceProperties* de la API de CUDA podem obtenir informació útil de la targeta gràfica com el model, la mida de la memòria compartida per bloc, el nombre de threads concurrents dins un bloc, el nombre màxim de threads que podem assignar a un bloc, el nombre màxim de blocs que podem crear en un kernel, la velocitat de la GPU, la mida de la memòria constant, si permet copies i execucions concurrents i el nombre de multiprocessadors que disposem.

```
Name:                          Tesla M2090
Total shared memory per block: 49152
Warp size:                     32
Maximum threads per block:     1024
Maximum dimension of grid:     65535
Clock rate:                    1301000
Total constant memory:         65536
Concurrent copy and execution: Yes
Number of multiprocessors:     16
```

En el Minotauro la comprovació d'errors en memòria ECC està desactivada, permet l'ús de memòria complet de 6GB i l'ample de banda màxim.

6 Paral·lelització del codi

6.1 Funcions objectiu

6.1.1 Anàlisi de les funcions

Tal com hem introduït a l'apartat 3.2.2, les dues funcions a paral·lelitzar són *calcEnergy* i *calcEnergyGradient* .

```
Funció calcEnergy ( e real[] coords, e NonBondPair[] nbi) retorna real energia
//Pre: coord i nbi estan inicialitzats
//Post: retorna l'energia acumulada de totes les intereccions contingudes en nbi

      per a cada nbi
                  energia := energia + onePairEnergy(nbi, coords)

retorna energia
```

Codi 5 Pseudocodi a alt nivell de calcEnergy.

En *calcEnergy* calculem l'energia de cada INC en la funció *onePairEnergy* i l'anem acumulant per obtenir l'energia final de la proteïna. Les successives crides a *onePairEnergy* no tenen ninguna dependència entre elles, i per tant, podem executar varies crides de *onePairEnergy* de manera concurrent, guardant per a cada una d'elles l'energia resultant de cada crida i després sumar-les totes en una reducció final per obtenir l'energia final de la proteïna.

```
Funció calcEnergyGradient ( e real[] coords, e NonBondPair[] nbis, s real[] grad)
retorna real energia
//Pre: coord, nbi i grad estan inicialitzats
//Post: retorna l'energia acumulada de totes les intereccions contingudes en nbi i el
gradient en el vector grad

      per a cada nbi en nbis
                  energia := energia + onePairEnergy( nbi, coords)
                  actualitzarGradient( grad,  nbi, coords)

retorna energia
```

Codi 6 Pseudocodi a alt nivell de calcEnergyGradient.

En la funció *calcEnergyGradient* a més de calcular l'energia també calculem el gradient de la INC (per estalviar-se el cridar dos cops les funcions comuns). El càlcul del gradient esta representat per la funció *actualitzarGradient* . El principal problema per a la paral·lelització d'aquesta part el trobem en el següent fragment de codi, de *actualitzarGradient* :

```
Funció actualitzarGradient( e/s real[] grad, e NonBondPair nbi, e real[] coords)
...
//es calcula el gradient i es desa el seu valor en les variables gradx, grady i gradz
...

grad[nbi->atomA.x] += gradx;
grad[nbi->atomA.y] += grady;
grad[nbi->atomA.z] += gradz;

grad[nbi->atomB.x] -= gradx;
grad[nbi->atomB.y] -= grady;
grad[nbi->atomB.z] -= gradz;
```

Codi 7 Actualització del vector grad en actualitzarGradient.

Aquesta funció primer calcularà el gradient per a una interacció en *gradx* , *grady* , *gradz* (que seràn els gradients per cada direcció). Llavors s'actualitza els valors del gradient de cada àtom de la INC en el vector *grad* (recordem que s'accedeix igual que el vector de coordenades[6]). Si volem executar concurrentment el càlcul del gradient, es pot donar la situació on un mateix àtom vulgui acumular els seus valors en el vector gradient en diferents fils d'execució i tindríem un conflicte de tipus “situació de competició”. Aquest conflicte podria donar lloc a una actualització perduda, ja sigui per una dependència de dades “Write after Read” o “Write after Write”.

6.2 CUDA

6.2.1 Metodologia

Per tal d'adaptar el codi a CUDA, es faran diverses iteracions on cada iteració es pot entendre com l'aplicació d'un seguit de transformacions al codi per resoldre els diversos problemes que van sorgint. Al finalitzar cada iteració es mostraran els resultats obtinguts mesurats utilitzant les tres funcions següents (codi a l'annex I.II.V):

- Funció *clock* de C per mesurar fragments de codi amb una precisió de 0,01s.
- Macro *TIMER* per mesurar funcions amb una precisió de 0,001s que utilitza les funcions de la 'Realtime library' de C.
- Esdeveniments CUDA per mesurar el temps de funcions a la GPU amb una precisió de 0,0001s.

En tot l'apartat 6, per tal de tenir valors fiables en la mesura dels temps d'execució, es realitzaran cinc proves i se'n calcularà la mitjana desestimant els “outlayers”.

En les dues primeres iteracions es treballa amb un primer joc de proves que conté una proteïna de 2.515 àtoms. En la tercera iteració introduïm un segon joc de proves que conté una proteïna de 31.313 àtoms.

6.2.2 Iteració 1

6.2.2.1 Objectiu

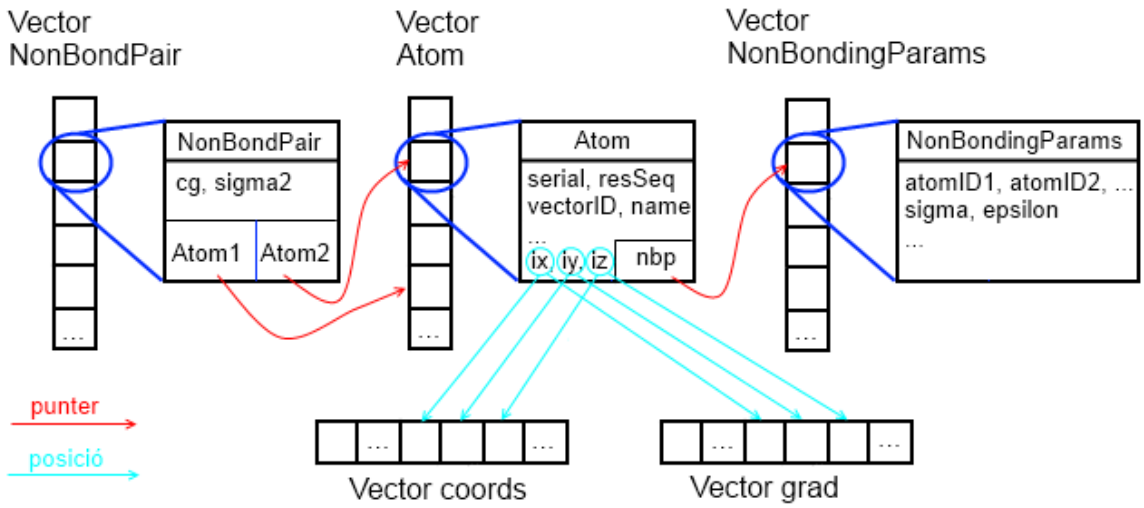
L'objectiu d'aquesta primera iteració és traslladar els càlculs de l'energia i el gradient a la GPU.

Crearem dos kernels, un pel càlcul de l'energia (*kernelcalcEnergy*) i un altre pel càlcul de l'energia i el gradient (*kernelcalcEnergyGradient*). Cada kernel s'executarà concurrentment en tants threads com INC existeixin i cada thread s'encarregara de realitzar els càlculs d'una INC.

6.2.2.2 Pas 1

En PELE++, s'utilitzen punters en les estructures de dades (veure figura 17). Aquests punters estan apuntant a zones de la memòria de la CPU i per tant deixaran de ser vàlids si movem les dades directament a la memòria de la GPU. Això ens obliga a fer un pre-procés per tal d'adaptar les dades i poder transferir-les a la GPU.

6 En l'apartat 3.2.2



Fig, 17 Estructures de dades en PELE++ utilitzades en les funcions calcEnergy i calcEnergyGradient

El primer pas és determinar les dades que necessitarem a la GPU per realitzar els càlculs de l'energia i el gradient de cada INC:

- De l'estructura *NonBondPair* els atributs *cg* , *sigma2* . Conté dos punters a estructures de tipus *Atom* .
- De l'estructura *Atom* els atributs *ix* , *iy* i *iz* serveixen per indicar les posicions on estan les coordenades de l'àtom en el vector *coords* . Els objectes de tipus *Atom* contenen un punter a una estructura de tipus *NonBondingParams* que conté algunes característiques electrostàtiques i de van der Waals.
- De l'estructura *NonBondingParams* l'atribut *epsilon* .
- El vector *coords* .

Aquestes serien les dades necessàries per realitzar els càlculs de l'energia i el gradient de cada INC. Per transferir-les a la GPU creem una nova estructura que anomenarem *NewNBP* . Cada INC queda representada per una d'aquestes noves estructures de dades. Les agruparem totes en un vector.

En el càlcul de l'energia i el gradient, només s'utilitza el vector *coords* per determinar la distancia entre els dos àtoms de la INC. Com passem la informació de cada interacció en el *NewNBP* , aprofitarem l'etapa de pre-procés per calcular aquesta distancia entre els àtoms. Així ens estalviem transferir a la GPU els atributs *ix* , *iy* , *iz* de cada àtom i el vector *coords* , aconseguint a més utilitzar menys memòria.

Els atributs de *NewNBP* seran:

- *cg* i *sigma2* provinents de *NonBondPair*
- *Aepsilon* i *Bepsilon* són l'atribut *epsilon* de *NonBondingParams* corresponent a cada àtom de la interacció (A per el primer, B per el segon)
- *dx* , *dy* i *dz* són les tres coordenades de la distancia entre els àtoms de la interacció, calculada en l'etapa de pre-procés

```
typedef struct{
	double cg, sigma2, Aepsilon, Bepsilon;
	float dx, dy, dz;
}NewNBP;
```

Codi 8 Contingut de l'estructura NewNBP,

La funció encarregada de crear el vector de *NewNBP* s'anomena *createCudaVector* .

6.2.2.3 Pas 2

El segon problema a resoldre en *calcEnergyGradient* és el problema dels accesos concurrents al vector gradient, ja que tenim un bucle on en cada iteració s'acumula el gradient dels dos àtoms de la INC en el vector gradient.

La solució proposada per poder realitzar el càlcul del gradient en la GPU, consisteix en guardar el càlcul del gradient de cada INC en un vector. Un cop els gradients de totes les INC estiguin calculats en aquest nou vector, se'n farà la reducció al vector *grad* en la CPU.

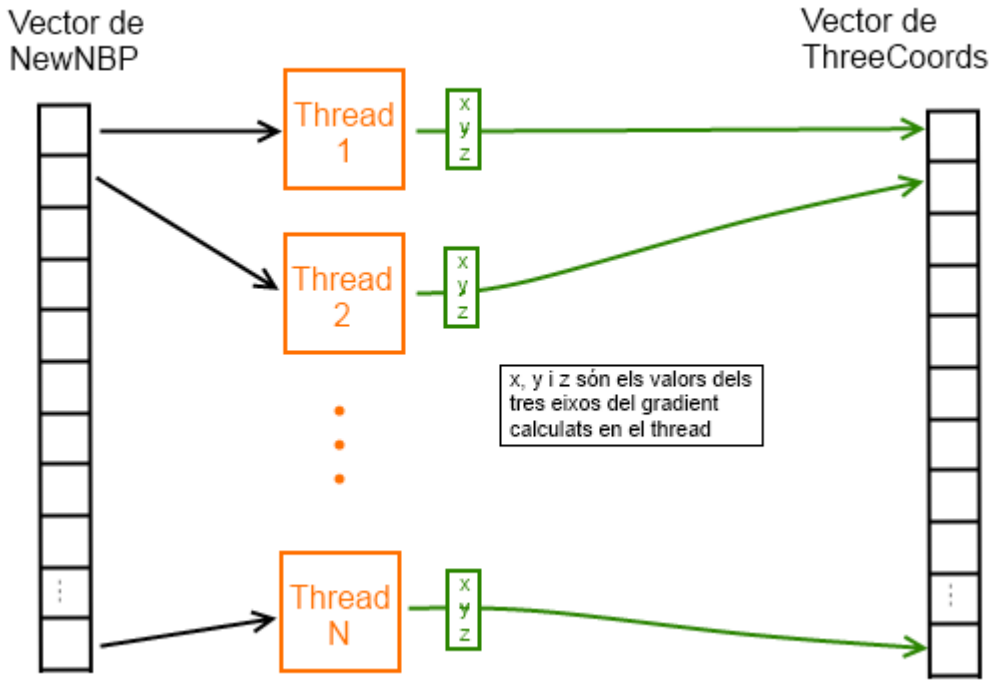


Fig. 18 Diagrama on cada thread obté la informació de la INC del vector de NewNBP, i desarà el gradient de la INC en el vector de ThreeCoords.

En el càlcul del gradient de cada INC, es calcula el gradient en l'eix x , y i z per a cadascun dels dos àtoms. Aquest gradient es desarà en un estructura anomenada *ThreeCoords* , formada per les tres coordenades x , y i z . El resultat del càlcul del gradient de la ièsima INC es desarà en la posició ièsima del vector de *ThreeCoords* . Un cop calculats tots els gradients de les INC, s'aplicarà la reducció final del gradient a la CPU.

```
typedef struct{
	double x, y, z;
}ThreeCoords;
```

Codi 9 Contingut de l'estructura ThreeCoords.

6.2.2.4 Pas 3

Un cop definides les dades, ja podem transportar els càlculs de les funcions de l'energia i gradient als seus respectius kernels amb les adaptacions oportunes per a la nova estructura de dades *NewNBP* , i en el cas del càlcul del gradient també *ThreeCoords* .

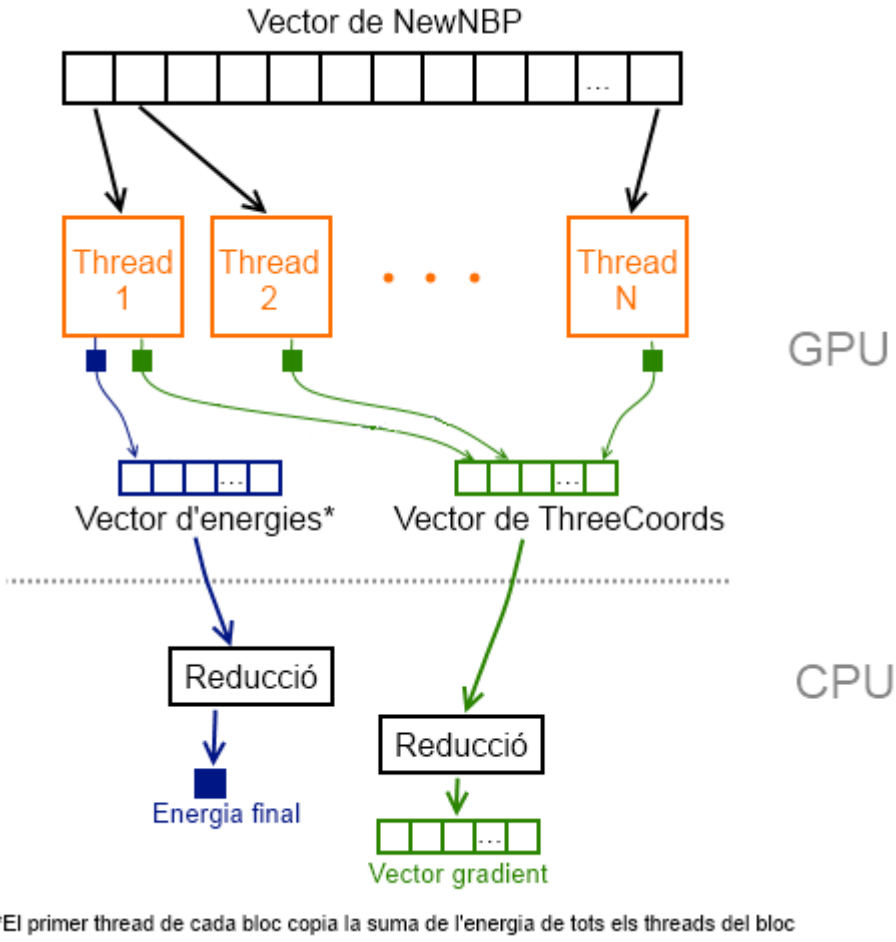


Fig. 19 Esquema de calcEnergyGradient.

Cada thread que executa qualsevol dels dos kernels, desarà en primer moment l'energia calculada en la memòria compartida entre els threads del bloc. Un cop tots els threads del bloc hagin calculat l'energia, faran una reducció per tal de tenir l'energia acumulada de tot el bloc. Així a la CPU s'enviara un vector amb les energies dels blocs (i no de tots els threads), i la reducció final per obtenir l'energia total de la proteïna serà més rapida.

Les funcions *calcEnergy* i *calcEnergyGradient* seran les encarregades de realitzar el pre-procés de les dades, copiar les dades de la memòria de la CPU a la GPU, executar els kernels, recollir les dades de la GPU cap a la CPU i fer les reduccions per tal d'obtenir l'energia i el gradient. Com a mostra, el codi de la funció *calcEnergyGradient* i el kernel *kernelcalcEnergyGradient* estan l'annex I.II.I.

6.2.2.5 Resultats

Al prendre mesures de les diferents etapes del codi hem observat que la primera crida a una funció CUDA, en el nostre cas *cudamalloc* , té un cost de 1,7s. Això es degut a la inicialització del sistema CUDA a la targeta gràfica. Aquest cost és constant i només passarà un cop, per aquesta raó s'ha tret dels resultats:

	calcEnergy	calcEnergyGradient
crear vector	0,065	0,065
cudaMalloc	0	0
cudaMemCpy	0,02	0,02
kernel	0	0
cudaMemCpy	0	0,02
reducció	0	0,03
TOTAL	0,085	0,135

(Temps en segons, 0s equival a un resultat menor a 0,01s)

Taula 4 Temps de calcEnergy i calcEnergyGradient després de la primera iteració.

Com podem observar, l'execució dels càlculs en la GPU són molt ràpids, on es gasta temps és en la creació del vector amb les dades, les copies a memòria i la reducció del gradient. El cost final és superior a la versió seqüencial, que tarda 0,045s per a *calcEnergy* i 0,087s per a *calcEnergyGradient* .

Observacions dels kernels:

- S'ha elegit el nombre de threads per bloc de la mida del warp, 32.
- El nombre de blocs va en funció del numero de threads per bloc i mida del vector d'INC. Per aquesta proteïna tenim suficients blocs disponibles (no superem els 65535).

6.2.3 Iteració 2

6.2.3.1 Objectiu

L'objectiu d'aquesta iteració és reduir el temps emprat en la creació del vector amb les dades per als kernels, i les copies entre la memòria de la CPU i GPU.

6.2.3.2 Pas 1

En l'anterior iteració, teníem totes les dades de les INC en un vector de *NewNBP* . Volem reduir la mida d'aquesta estructura per reduir el temps que tarda en copiar-se en memòria.

En cada *NewNBP* tenim informació dels dos àtoms que formen la INC. Com que cada àtom apareix en moltes INC, estem replicant la mateixa informació en cada *NewNBP* . Per evitar-ho crearem un vector amb la informació de tots els àtoms i en el *NewNBP* tindrem la posició dels dos àtoms al vector d'àtoms (evitant així l'ús de punters). També eliminarem els atributs que feien referencia a la distancia entre els àtoms de la INC i farem que la distancia sigui calculada en els kernels i no en l'etapa de pre-procés. Per poder realitzar el càlcul de la distancia haurem de passar el vector de les coordenades dels àtoms a la GPU.

La nova estructura de *NewNBP* es la següent:

```
typedef struct{
      double cg, sigma2;
      int atomA, atomB;
}NewNBP;
```

Codi 10 Nou contingut de l'estructura NewNBP.

La nova estructura que contindrà la informació necessària de cada àtom, i en crearem un vector, és la següent:

```
typedef struct{
      double epsilon;
      unsigned int ix, iz, iy;
}NewAtom;
```

Codi 11 Contingut de l'estructura NewAtom.

Un cop definides les noves estructures que utilitzarem en els kernels, ja els podem modificar.

El codi del kernel modificat *kernelcalcEnergyGradient* està a l'annex I.II.II.

6.2.3.3 Pas 2

En PELE++, les INC dels àtoms no sempre canvien abans de cridar a *calcEnergy* i *calcEnergyGradient*. Per aquesta raó treurem el pre-procés d'adaptació de les dades de dins les funcions *calcEnergy* i *calcEnergyGradient*.

Les especificacions de les funcions de pre-procés i d'alliberar la memòria de la GPU creades són les següents:

```
void createAtomVector()
//Pre: existeix el vector d'atoms en PELE
//Post: crea el vector d'atoms per CUDA i el copia a la memoria de la targeta grafica

void createNBIVector( vector< NonBondPair > & nonBondingInteractions)
//Pre: existeix el vector d'INC en PELE
//Post: crea el vector de parelles d'atoms per CUDA i el copia a la memoria de la
targeta grafica

void updateNBIVector( vector< NonBondPair > & nonBondingInteractions)
//Pre: vector d'INC previament creat
//Post: actualitza el vector de parelles de la memoria de la targeta grafica

void copyCoordsToGPU(double *coords)
//Pre: existeix el vector de coordenades
//Post: copia el vector de coordenades a la memoria de la targeta grafica

void freeAtomVector()
//Pre: -
//Post: allibera la memoria del vector d'atoms de la targeta grafica

void freeNBIVector()
//Pre: -
//Post: allibera la memoria del vector d'atoms de la targeta grafica

void freeCoordsGPU()
//Pre: -
//Post: allibera la memoria del vector d'atoms de la targeta grafica
```

Codi 12 Especificacions de les funcions de pre-procés.

Les dades hauran d'estar preparades i en la GPU abans d'utilitzar *calcEnergy* i *calcEnergyGradient*. El codi de *createAtomVector*, *createNBIVector* i *copyCoordsToGPU* estan a l'annex I.II.II.

6.2.3.4 Pas 3

Les copies de dades de la memòria principal a la memòria de la GPU es poden realitzar de manera asíncrona, així la CPU pot seguir treballant. Per tant substituirem la funció *cudaMemcpy* per *cudaMemcpyAsync* on sigui possible i reordenarem el codi per tal de maximitzar l'ús de la CPU i GPU simultàniament.

Com les crides als kernels també són asíncrones, una altra modificació serà la col·locació d'instruccions que no siguin crides síncrones de CUDA després de la crida al kernel. Un exemple seria:

```
...
kernelcalcEnergyGradient<<<num_blocks,THREADS_PER_BLOCK>>>(dev_vnbp,dev_vatom,
dev_coords, dev_numPairs, dev_result, dev_tcoords);
tcoords=(ThreeCoords*)malloc(numPairs*sizeof(ThreeCoords));
result=(double*)malloc(num_blocks*sizeof(double));
cudaMemcpy(tcoords, dev_tcoords, numPairs*sizeof(ThreeCoords), cudaMemcpyDeviceToHost);
...
```

Codi 13 Fragment de codi de calcEnergyGradient, on es mostra en color verd la col·locació de funcions entre llançament del kernel i la següent funció CUDA.

6.2.3.5 Resultats

El fet d'utilitzar les noves estructures en el pas 1, ha millorat el rendiment reduint el temps en la creació dels vectors i en la copia de les dades a memòria. El temps d'execució de *calcEnergyGradient* ara és de 0,09s i ja s'aproxima als 0,087s de la versió seqüencial.

	calcEnergyGradient iteració 2	calcEnergyGradient iteració 1
crear vectors	0,03	0,065
cudaMalloc	0	0
cudaMemCpy	0,01	0,02
kernel	0	0
cudaMemCpy	0,02	0,02
reducció	0,03	0,03
TOTAL	0,09	0,135

(Temps en segons, 0s equival a un resultat menor a 0,01s)

Taula 5 Comparativa entre el temps de calcEnergyGradient de la iteració 1 amb el de la iteració 2.

En el pas 2, al treure les funcions del pre-procés de *calcEnergyGradient* hem reduït el temps de la funció passant de 0,087s en el codi seqüencial a 0,045s.

	Temps en la iteració 2
createAtomCudaVector()	0
createNBICudaVector()	0,042
copyCoordsToGPU()	0
calcEnergyGradient()	0,045

(Temps en segons, 0s equival a un resultat menor a 0,001s)

Taula 6 Temps de les funcions de l'etapa de pre-procés i de calcEnergyGradient sense els càlculs de pre-procés al finalitzar la iteració 2.

6.2.4 Iteració 3

6.2.4.1 Objectiu

El primer objectiu d'aquesta iteració es adaptar el codi per tal de poder calcular l'energia i el gradient de proteïnes de qualsevol mida; en el desenvolupament fins ara estavem limitats pel nombre de threads i blocs màxims de CUDA.

El segon objectiu és reduir el temps emprat en el càlcul del gradient.

6.2.4.2 Pas 1

En CUDA hi ha un número màxim de blocs i threads. En el nostre codi, els threads queden fixats per la constant *THREADS_PER_BLOCK*, i el número de blocs a utilitzar el calculem a partir del

nombre d'INC que tenim i el número de threads per bloc. D'aquesta manera cada thread tracta una interacció.

```
num_blocks = ceil((double)numInteractions  / (double) THREADS_PER_BLOCK);
```

Codi 14 Instrucció que calcula els nombre de blocs a utilitzar.

La proteïna 1 utilitzada fins ara per provar i mesurar el temps dels canvis fets, no sobrepassa el màxim de blocs permesos en CUDA que són 65535 i no hem tingut cap problema. Ara volem adaptar el codi per poder calcular proteïnes amb un nombre arbitrari d'INC. El canvi que s'ha introduït ha sigut fer que cada thread tracti més d'una interacció si es necessari.

La implementació és senzilla, ja que nomes ha fet falta canviar la condició que donava dret al thread a fer el càlcul, per un bucle que s'incrementa pel nombre total de threads en el grid, i anar acumulant l'energia en el propi thread.

Una parella per thread:

```
__global__ void kernelcalcEnergy( NewNBP *vNBP, NewAtom *vAtom , double *coords, int
*num, double *result)
{

	int tid = threadIdx.x + blockIdx.x * blockDim.x;
	...

	if( tid<num_interactions )
	{
		...
		res[buff_index]=electrostatic+lennard_jones;
	}
	...
}
```

Codi 15 Extracte de kernelcalcEnergy on un thread calcula l'energia d'una INC.

Més d'una parella per thread:

```
__global__ void kernelcalcEnergy( NewNBP *vNBP, NewAtom *vAtom , double *coords, int
*num, double *result)
{

	int tid = threadIdx.x + blockIdx.x * blockDim.x;
	...

	while( tid<num_interactions )
	{
		...
		res[buff_index]+=electrostatic+lennard_jones;
		tid+=gridDim.x*blockDim.x;
	}
	...
}
```

Codi 16 Extracte de kernelcalcEnergy on un thread calcula l'energia de més d'una INC.

Tots els kernels que tractin amb àtoms i INC es programaran d'aquesta forma, permetent que cada thread pugui tractar més d'un àtom o INC.

6.2.4.3 Pas 2

En el codi de la reducció del gradient a la CPU (veure codi 17), s'observa una dependència de dades entre iteracions del bucle, degut a la no coneixença de la posició del vector *grad* que hem d'actualitzar en cada instrucció perquè la posició és determinada per cada àtom.

```
for(i=0; i<numPairs;i++){
            grad[ nonBondingInteractions[i].first->ix ] += tcoords[i].x;
            grad[ nonBondingInteractions[i].first->iy ] += tcoords[i].y;
            grad[ nonBondingInteractions[i].first->iz ] += tcoords[i].z;
            grad[ nonBondingInteractions[i].second->ix ] -= tcoords[i].x;
            grad[ nonBondingInteractions[i].second->iy ] -= tcoords[i].y;
            grad[ nonBondingInteractions[i].second->iz ] -= tcoords[i].z;
}
```

Codi 17 Reducció del gradient a la CPU. Es redueix el vector tcoords en el vector grad.

Al fer el càlcul d'una INC, trobarem dos gradients parcials, el degut al primer àtom, i el degut al segon àtom. Aquests dos gradients parcials són iguals però de signe diferent (estan representant dues forces oposades). Per tant, només cal fer un dels càlculs i acumular en el vector *grad* el gradient parcial del primer àtom, i el gradient parcial negatiu degut al segon àtom. Per aquesta raó en $tcoords[i]$ només es guarda el gradient parcial positiu per la INC ièsima ($nonBondingInteractions[i]$) .

Per poder millorar el temps de la reducció provarem d'executar-la també a la GPU. Per no tenir conflictes en l'accés al vector *grad* , cada thread s'encarregarà de reduir el gradient d'un àtom, es a dir, el thread acumularà en el vector *grad* tots els gradients generats en les INC en les quals l'àtom forma part. Això ens permetrà realitzar la reducció del gradient en paral·lel, evitant els conflictes que apareixerien si ens limitéssim a repartir les iteracions del bucle anterior (codi 17) en diferents threads.

Per aconseguir que cada thread pugui acumular el gradient d'un únic l'àtom, tenim que trobar els gradients de les seves INC en el vector *tcoords* . Això ho resoldrem amb una taula de mapeig que ens indicara per cada àtom on té gradients dins *tcoords* .

Aquesta taula té les següents característiques:

- Dues columnes, la primera conté l'identificador de l'àtom i la segona la posició en el vector *tcoords* on hi ha un gradient degut a la participació d'aquest àtom en una interacció.
- Les mateixes files que número de INC, donat que a cada estructura *tcoords* es guarda el gradient resultant de cada INC.
- Un àtom apareix en tantes files com INC tingui, però sempre amb una posició del vector *tcoords* diferent.
- Estarà ordenada per la columna de l'identificador dels àtoms, per així tenir agrupats els àtoms i facilitar l'accés al thread que en vulgui reduir el gradient.

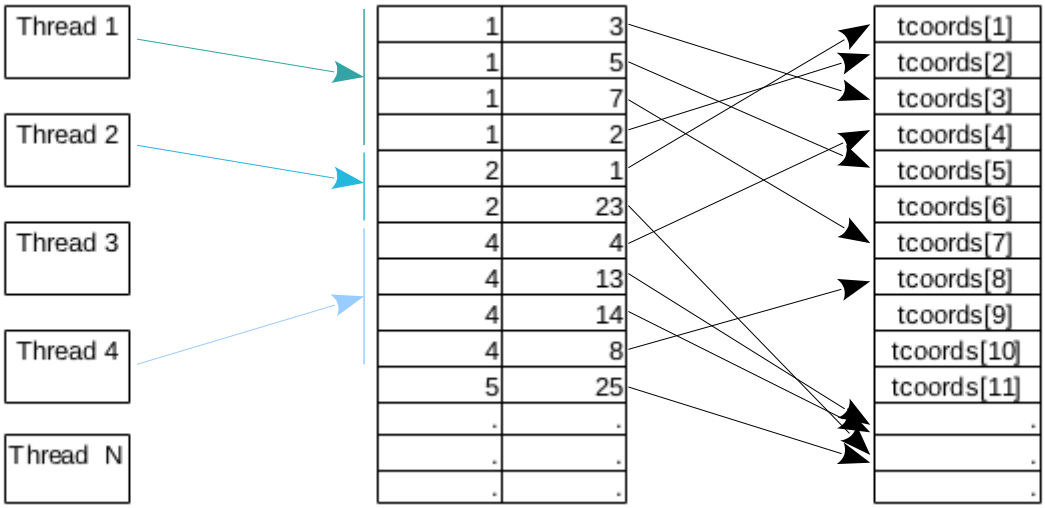


Fig. 20 Exemple de com una taula de mapeig permet al thread 1 accedir a tots els gradients emmagatzemats en el vector tcoords de l'àtom 1, al thread 2 els de l'àtom 2, etc.

Com una INC està formada per dos àtoms, crearem una taula pels primers àtoms de les INCs i una altra taula pels segons àtoms. La raó es que al acumular els gradients, el primer àtom una suma al vector *grad* mentre que el segon àtom una resta (es pot observar en el codi de la codi 17).

Els passos per implementar aquestes taules i portar a terme la reducció del gradient a la GPU són:

1. Crear les dues taules de mapeig
2. Ordenar les taules de mapeig per àtoms
3. Crear vector amb la posició de la taula on comença cada àtom (vector d'inici)
4. Fer la reducció del gradient

Els primers 3 passos només es fan quan es modifiquen les INC, en l'etapa de pre-procés. La funció encarregada és *createMapCudaVectors* a l'annex I.II.III.

Tots els passos els realitzarem en la GPU. D'aquesta manera en tot moment les dades estan en la memòria de la targeta gràfica i ens estalviem transferències de memòria de la CPU a la GPU i viceversa.

Per generar les dues taules a la GPU, cada thread tractarà una INC i escriurà a les dues taules els identificadors dels àtoms i quina INC és (així podem accedir al gradient desat en el vector *tcoords*). El kernel encarregat de generar les taules és *kernelCreateMaps* i el codi està a l'annex I.II.III.

Cada taula estarà formada formada per dos vectors que representaran les columnes.

Un cop les taules s'han generat, hem d'ordenar-les per la columna dels àtoms. Per realitzar l'ordenació de la forma més ràpida, utilitzem la biblioteca Thrust [7] que ens aporta una funció d'ordenació *sort_by_key* (utilitza l'algorisme Radix sort). A la funció se li passen dos vectors: un que contindrà les claus que hem d'ordenar, i un altre que conté els valor que corresponen a cada clau. És a dir, és com si tinguéssim una matriu amb dues columnes (clau i valor) i els ordenem per la columna de la clau. La nostra clau és l'identificador de l'àtom.

Al utilitzar la biblioteca Thrust, primer hem de transformar les nostres variables en un format especial per la funció *sort_by_key*. Amb el *device_ptr* s'indica que estem treballant amb posicions de memòria de la GPU.

```
thrust::device_ptr<int> dev_ptr1k(dev_map1k);
thrust::device_ptr<int> dev_ptr1v(dev_map1v);
thrust::device_ptr<int> dev_ptr2k(dev_map2k);
thrust::device_ptr<int> dev_ptr2v(dev_map2v);
```

Codi 18 Adequació de les dades per ser utilitzades en les funcions de la biblioteca Thrust. En aquest cas transformem les dues taules de mapeig, formades cada una d'elles per dos vectors (clau i valor).

Ara ja podem cridar la funció, on el primer camp es l'inici del vector amb el numero de l'àtom, el segon el final del vector i l'últim el vector amb les posicions de l'àtom en la llista de parelles.

```
thrust::sort_by_key( dev_ptr1k, dev_ptr1k + numInteractions, dev_ptr1v);
thrust::sort_by_key( dev_ptr2k, dev_ptr2k + numInteractions, dev_ptr2v);
```

Codi 19 Crida a la funció d'ordenació sort_by_key de la biblioteca Thrust.

Quan tenim les dues taules ordenades, s'ha de crear un vector per a cada taula que ens indicarà on està la primera fila de cada àtom dins la taula. Aquests vectors els anomenarem vectors inici on en la posició ièsima tindran la primera de l'àtom ièsim en la taula de mapeig.

El kernel encarregat de crear aquest vector d'inici és *kernelcreateStart* i està a l'annex I.II.III. En aquest kernel, cada thread s'encarregarà d'un àtom i buscarà la posició on comença aquest àtom dins la taula de mapeig ordenada. S'utilitzarà la cerca dicotòmica al ser l'algorisme de cerca més ràpid. Com a dins el mapa ens podem trobar repeticions d'àtoms, la cerca dicotòmica no ens assegura que la posició trobada sigui la primera.

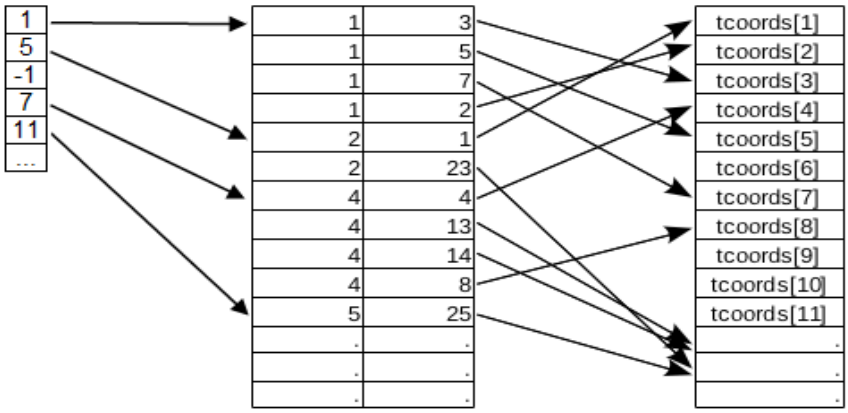


Fig. 21 Imatge del vector d'inici senyalant la primera posició dels àtoms en la taula de mapeig i com aquesta senyala la posició del gradient de l'àtom en el vector tcoords.

Per aquesta raó, després de realitzar la cerca dicotòmica, si hem trobat l'àtom busquem la posició del seu primer element. Si no existeix l'àtom marquem la posició en el vector d'inici amb un -1.

Un cop tenim totes les estructures preparades ja podem realitzar la reducció del gradient a la GPU. El kernel encarregat de fer la reducció és *kernelgradReduction* a l'annex I.II.III.

En la reducció, el càlcul del gradient total de cada àtom estarà assignat a un únic thread, que serà l'encarregat de recórrer les files de les dues taules de mapeig de l'àtom per accedir als gradients de les seves INC, que estan desats en el vector *tcoords* . Utilitzem variables temporals per acumular el gradient dins el thread i al final el copiarem en el vector gradient. Així minimitzem els accessos a memòria principal.

En el kernel tenim dos bucles que recorren les files de les dues taules de mapeig, guardades en la memòria global de la GPU. Cada bucle, en cada iteració, comprova si encara està tractant el mateix àtom, ja que el bucle finalitza quan comencem a tractar un altre (per exemple, en el procés mostrat en la figura 21, si un thread està tractant l'àtom 1, finalitza quan el identificador de l'àtom passa a

ser 2). Si coneixéssim la posició final no faria falta accedir cada vegada a la taula i ens estalviarem molts accessos a la memòria global (molt lents), per tant la calcularem.

El rendiment d'aquest algorisme dependrà bastant dels vectors d'inici i del hardware, ja que introduïm instruccions de salt i un bucle. Com no sabem quina opció és millor (calcular o no l'últim element), s'han implementat dues funcions de reducció per poder experimentar i veure que penalitza més, si l'accés a memòria o les noves instruccions introduïdes. El *kernelgradReduction2* implementa aquesta modificació a l'annex I.II.III.

El codi de *calcEnergyGradient* , modificat per realitzar la reducció del gradient a la GPU, està a l'annex I.II.III.

6.2.4.4 Resultats

Els resultats d'aquesta tercera iteració s'han desglossat en funcions i en el cas de *calcEnergyGradient* hem detallat el temps dels dos kernels que conté. També hem introduit un nova proteïna de mida superior per comprovar el correcte funcionament i veure el seu comportament.

	Proteina 1	Proteina 2
createAtomCudaVector()	0,001	0,001
createNBICudaVector()	0,042	0,091
createMapCudaVectors()	0,008	0,015
copyCoordsToGPU()	0	0,001
calcEnergyGradient() (temps total)	0,0130	0,0240
kernelcalcEnergyGradient()	0,0023	0,0040
kernelgradReduction()	0,0088	0,0172
calcEnergyGradient() (temps total)	0,0100	0,0150
kernelcalcEnergyGradient()	0,0023	0,0040
kernelgradReduction2()	0,0058	0,0087

(Temps en segons, 0s equival a un resultat menor a 0,001s)

Taula 7 Temps de les execucions de les funcions utilitzant dos jocs de proves. Per a la funció calcEnergyGradient hi ha dues versions segons el kernel de la reducció del gradient utilitzat.

El nostre codi ja accepta proteïnes de qualsevol mida ja que la proteïna 2 té més INC que blocs de 32 threads podem crear.

En referencia al rendiment, en la funció *calcEnergyGradient* , podem observar com el resultat d'utilitzar la funció *kernelgradReduction2* ha sigut millor que utilitzant *kernelgradRecution* en els dos casos. Per tant, podem arribar a la conclusió que penalitza més accedir cada vegada a la memòria global per comprovar si ja hem arribat al final dels elements de l'àtom en el vector, que no pas calcular la posició final de l'àtom a l'inici.

Finalment, gràcies al càlcul del gradient a la GPU, la funció *calcEnergyGradient* ha millorat el rendiment aconseguit a la segona iteració amb la proteïna 1 passant de 0,045s a 0,01s. També hi ha una nova funció en l'etapa de pre-procés, però el seu cost és de 0,008s que és 4,3 vegades inferior al temps que em guanyat.

Al finalitzar aquesta tercera iteració, la millora de temps en la funció *calcEnergyGradient* entre el codi seqüencial inicial i el codi CUDA és:

- proteïna 1: 0,087→0,01
- proteïna 2: 0,181→0,015

En la figura 22 es mostra un diagrama amb les funcions que s'executen a la CPU i a la GPU.

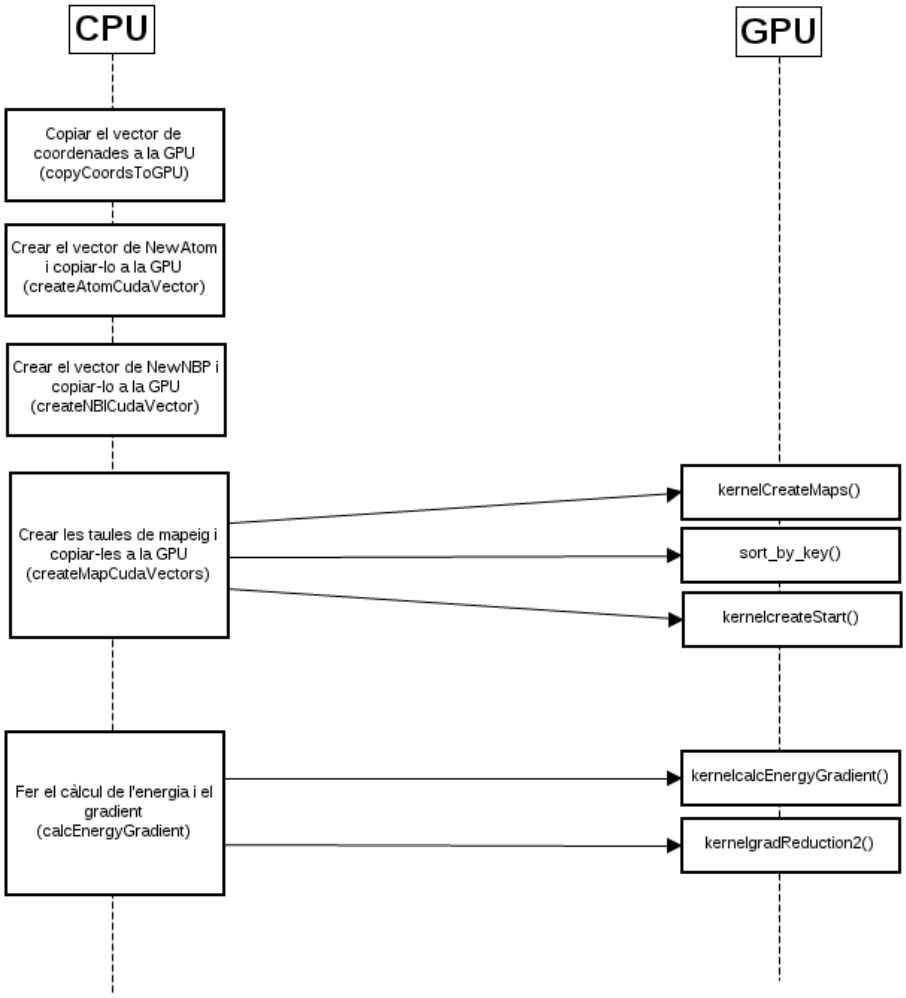


Fig. 22 Diagrama de les funcions que es realitzen en la CPU i en la GPU per tal de realitzar el càlcul de l'energia i el gradient.

6.3 OpenCL

La versió d'OpenCL serà una traducció de la millor versió implementada en CUDA, amb l'excepció de l'ús de la funció d'ordenació sort de la biblioteca Thrust de CUDA. Al no existir cap biblioteca “oficial” amb una funció d'ordenació en OpenCL s'ha hagut d'implementar.

A continuació es mostren les dues principals diferencies, la inicialització d'OpenCL i la funció d'ordenació.

6.3.1 Inicialització

Primer de tot s'ha d'inicialitzar el sistema tal i com ja s'ha explicat a l'apartat 4.3 del document.[7]

El codi està a l'annex I.II.IV.

6.3.2 Funció d'ordenar vectors

En el codi CUDA, s'utilitza la funció sort de la biblioteca Thrust que implementa l'algorisme d'ordenació Radix sort. Es va realitzar una cerca per tal de trobar alguna biblioteca o codi que implementes Radix sort i fos possible aprofitar en el nostre projecte.

Els resultats de la cerca van ser els següents:

- Biblioteques:
  - Clogs[8]: Es una biblioteca d'alt nivell que implementa dues funcions en OpenCL, Radix sort i Exclusive Scan. El problema va ser el no poder instal·lar-la i provar al clúster Minotauro.
  - Clpp[9]: Es una biblioteca que esta en fase beta i no hi ha documentació.
- Codis:
  - “A portable implementation of the Radix sort algorithm in OpenCL”[10]: L'algorisme no complia tots els requisits, només ordena les claus i no els valors. Es va modificar perquè ordenes els valors però donava errors i no funcionava per tots els jocs de prova.

Vist el poc èxit i la dificultat d'utilitzar algunes de les solucions anteriors, es va decidir implementar des de cero un algorisme d'ordenació. Així ens assegurem del seu correcte funcionament, el codi és més compacte i es possible reutilitzar-lo més endavant en altres projectes.

L'algorisme elegit és el Bitonic sort, ja que s'adapta bé a la paral·lelització i sembla més senzill d'implementar que el Radix sort.

6.3.2.1 Bitonic sort

El Bitonic sort és un dels algorismes més ràpids en ordenar “sorting networks”. Les “sorting networks” són algorismes de cerca que no tenen dependència de dades entre les seqüencies de comparacions. Per aquesta raó s'adapta bé als sistemes multiprocessador [8].

L'algorisme es basa en tenir seqüencies d'elements ordenats donada una entrada. El nombre d'elements d'aquestes seqüencies creixera de forma exponencial en base 2, començant en 2 i acabant amb la totalitat dels elements. L'ordenació de les seqüencies es divideix en etapes on es realitzaran

7 S'ha pres de referència la inicialització del llibre OpenCL Programming Guide
8 http://clogs.sourceforge.net
9 http://code.google.com/p/clpp/
10 http://code.google.com/p/ocl-radix-sort/

les comparacions necessàries entre parelles d'elements per tal de tenir la seqüencia ordenada. La figura 23 il·lustra l'algorisme de manera gràfica.

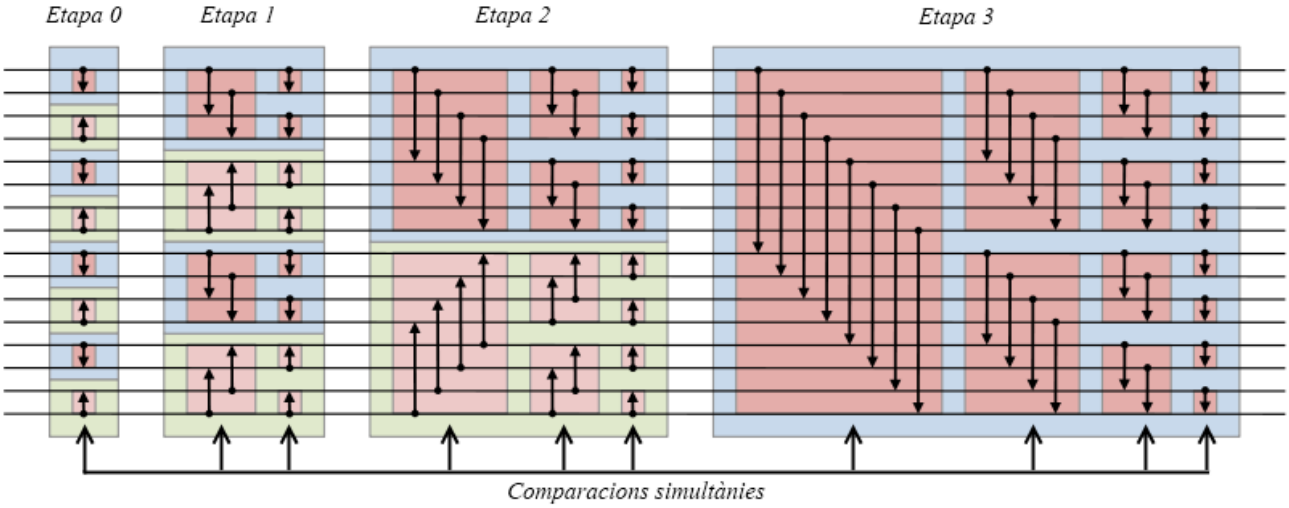


Fig. 23 Algorisme Bitonic sort.

Per una entrada de 2^n elements, tindrem $\log(n)$ etapes[11], on en cada etapa x tindrem grups de $2^{n/2x}$ elements ordenats.
Basant-nos en la figura 23, tenim 4 etapes ($\log(16)=4$). Els quadrats de color blau indiquen que l'ordre dels elements al finalitzar l'etapa ha de ser ascendent, i en els quadrats de color verd ha de ser descendent. Els comparadors són els requadres de color vermell i engloben grups de comparacions que es poden realitzar de forma simultània. El número de comparacions simultànies és $(\log(n)\cdot(\log(n)+1))/2$.

El cost de l'algorisme és $n\cdot\log(n)^2$, però al executar-lo de forma paral·lela en n processadors passa a ser $O\log(n)^2$ en tots els casos.

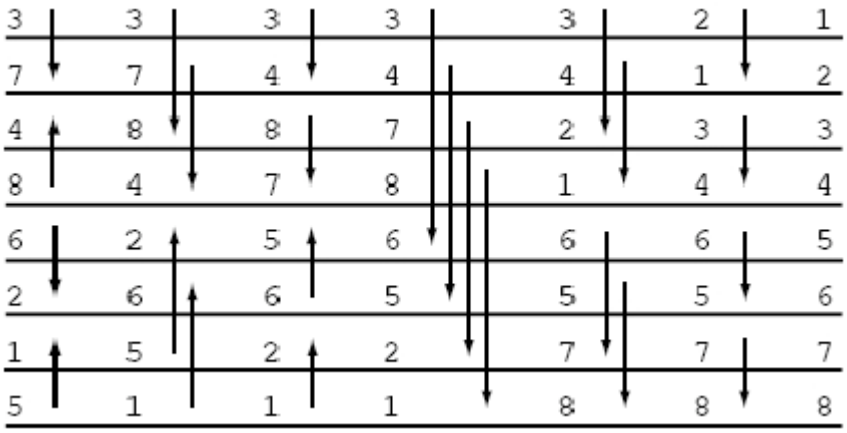


Fig. 24 Exemple de Bitonic sort per una entrada de 8 elements.

6.3.2.2 Implementació

S'han creat 3 versions del Bitonic sort, la següent és la que millor rendiment ha donat. En l'annex queden recollides les altres dues implementacions.

11 Tots els logaritmes en base 2

L'algorisme necessita que el nombre d'elements d'entrada sigui potencia de 2, per aquesta raó si fos necessari s'afegeix padding per satisfer aquesta condició.

La part que podem implementar en la GPU són les comparacions que es poden realitzar de manera simultània. Per aquesta raó assignem un thread a cada comparació. Si tenim n elements el nombre de comparacions i threads és $n/2$.

Alhora d'implementar el codi de la GPU, es va veure que depenem del treball d'altres threads quan anem avançant entre comparacions simultànies. Hi ha una dependència “Read after Write”. Com que només podem sincronitzar threads d'un mateix bloc i no threads de blocs diferents, hem optat per llençar un kernel (anomenat *bitonic_simple_v3*) per a cada comparació simultània. Per aquesta raó dins la funció que llença els kernels, tenim un primer bucle que va iterant sobre les etapes de l'algorisme, i dins seu un segon bucle que itera sobre les comparacions simultànies dins a cada etapa cridant el kernel que compara els elements.

Una optimització que s'ha aplicat ha sigut la creació d'un kernel (anomenat *bitonic_multi_v3*) encarregat de realitzar diverses comparacions simultànies quan tots els elements implicats estiguin al mateix bloc, i per tant no hi ha dependència de dades entre blocs i es poden sincronitzar el threads de dins el bloc. Aquesta funció tractarà tantes comparacions com threads per bloc hi hagi.

El codi de la funció que implementa l'algorisme Bitonic sort i el codi dels dos kernels és a l'annex I.II.IV.

En el codi dels kernels ens aprofitem de les propietats binaries de les potencies de 2 per coneixer la direcció de la comparació i si es el primer element de la comparació. En el kernel tenim la variable dir, que indica la direcció de la comparació (1 si és ascendent o 0 si es descendent), i la variable en, que indica si l'element és el primer element de la comparació (si val 0 és el primer element). Els threads que realitzen les comparacions i intercanvien els elements de la parella comparada, sempre s'assignaran al primer element de la comparació.

bit 3	bit 2	bit 1	bit 0	Thread
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

dir en

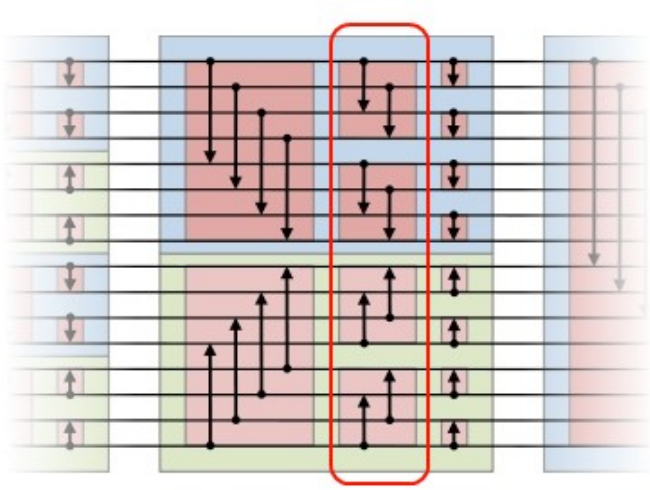


Fig. 25 Exemple dels valors que prenen les variables dir i en en la comparació 1 de l'etapa 2.

6.3.2.3 Resultats

Per poder trobar el punt òptim de rendiment, s'ha provat l'algorisme modificant el nombre de

threads per bloc. En l'experimentació s'ha utilitzat una proteïna amb 4.239.271 interaccions per mesurar el temps emprat en l'ordenació dels dos vectors de mapes.

Primer de tot els resultats obtinguts només utilitzant la funció *bitonic_sort_simple_v3*:

Threads per bloc	Temps (s)
32	0,70
64	0,47
128	0,41
256	0,42

Taula 8 Resultats obtinguts utilitzant bitonic_sort_simple_v3 per realitzar tota la ordenació.

Els resultats utilitzant també la funció *bitonic_sort_multi_v3* són els següents:

Threads per bloc	Temps (s)
32	0,63
64	0,40
128	0,34
256	0,34

Taula 9 Resultats obtinguts utilitzant bitonic_sort_simple_v3 i bitonic_sort_multi_v3.

Tal i com es pot comprovar, l'ús de *bitonic_sort_multi_v3* millora els resultats finals.

El nombre ideal de threads per bloc sembla ser 128 o 256. Per assegurar-nos, hem creat un benchmark per tal de mesurar quan temps necessita l'algorisme per ordenar un vector de 2^n elements modificant el nombre de threads per bloc. El màxim nombre d'elements que hem pogut provar ha estat 2^{26}, ja que per 2^{27} elements ens ha donat error de memòria.

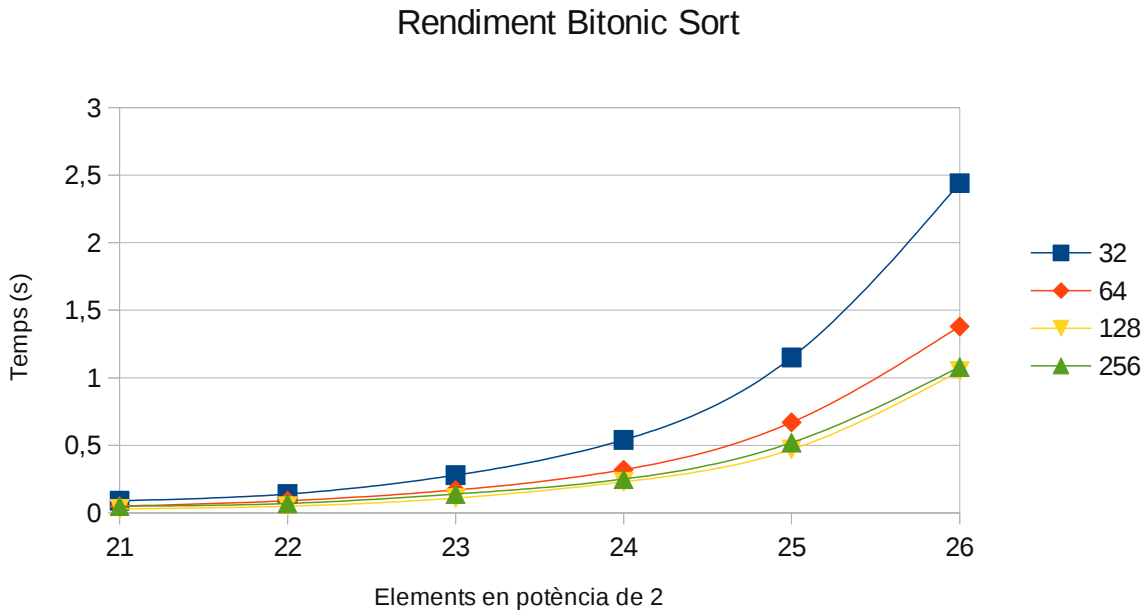


Fig. 26 Gràfic amb el resultat del benchmark variant el nombre de threads per bloc i el nombre d'elements a ordenar.

Els resultats (veure figura 26) ens mostren com el millor rendiment el trobem al usar 128 threads, i és d'uns 64 milions d'elements per segon. Com s'observa en el gràfic de la figura 26, quan doblem el nombre d'elements a ordenar es dobla el temps necessari per ordenar-los.

El rendiment del Radix sort de CUDA és de 775 milions d'elements per segon en una Geforce GTX480 [9], que té una potencia similar a la M2090 del Minotauro (1,3 TFLOPs les dues). Aquesta diferencia tan gran era d'esperar ja que la biblioteca Thrust de CUDA conté funcions d'alt rendiment i nosaltres hem creat una implementació sense gaires optimitzacions.

Un dels problemes del Bitonic sort és la necessitat de que l'entrada sigui potencia de dos, fet que em resolt afegint padding, però que ens penalitza el rendiment.

6.3.3 Resultats implementació en OpenCL en comparació amb CUDA

Utilitzant les mateixes proteïnes que en l'apartat de CUDA, els resultats són els següents:

	CUDA		OpenCL	
	Proteina 1	Proteina 2	Proteina 1	Proteina 2
createAtomVector()	0,001	0,001	0	0,002
createNBIVector()	0,042	0,091	0,057	0,119
createMapVectors()	0,008	0,015	0,088	0,184
copyCoordsToGPU()	0	0,001	0	0
calcEnergyGradient()	0,01	0,015	0,01	0,02

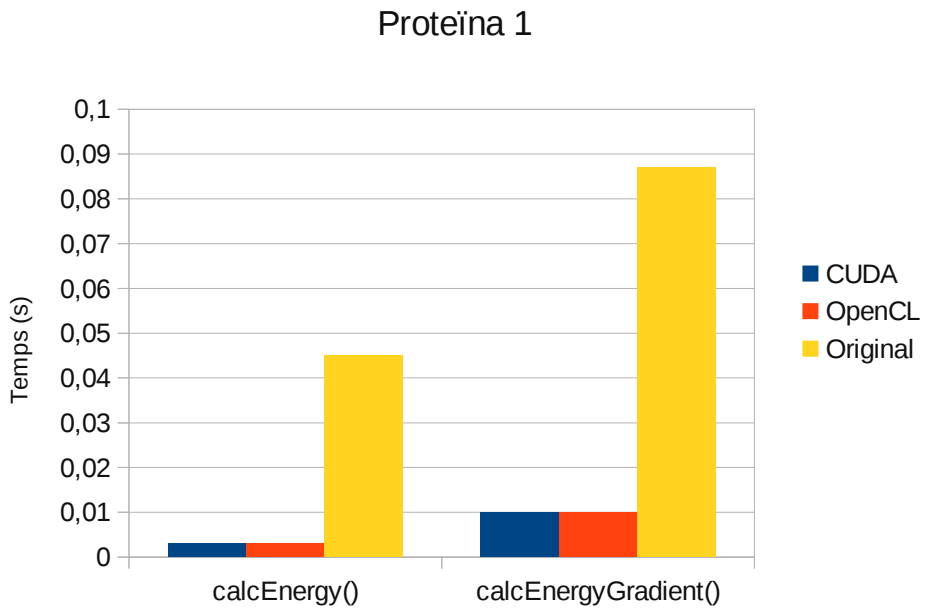
(Temps en segons, 0s equival a un resultat menor a 0,001s)

Taula 10 Comparativa de temps entre les funcions de la versió CUDA i OpenCL.

Podem observar com el rendiment de la versió CUDA és lleugerament superior a la versió en OpenCL, amb excepció de la funció *createMapCudaVectors* , ja que la versió d'OpenCL conté la funció d'ordenació implementada per nosaltres i és molt més lenta que la funció implementada en la biblioteca de CUDA.

6.4 Conclusions

Hem aconseguit millorar el temps de les dues funcions del càlcul de l'energia i el gradient, però hem introduït noves funcions per transformar les dades i poder fer els càlculs a la GPU. Com aquestes funcions noves s'han d'utilitzar abans de calcular l'energia i el gradient, farem un model de PELE integrant les noves funcions per veure'n l'ús que en fa, i així calcular el guany real que suposaria l'aplicació de CUDA i OpenCL. Tot i això els temps aconseguits per a les funcions *calcEnergy* i *calcEnergyGradient* , i el seu speed up respecte al codi original, són els següents:

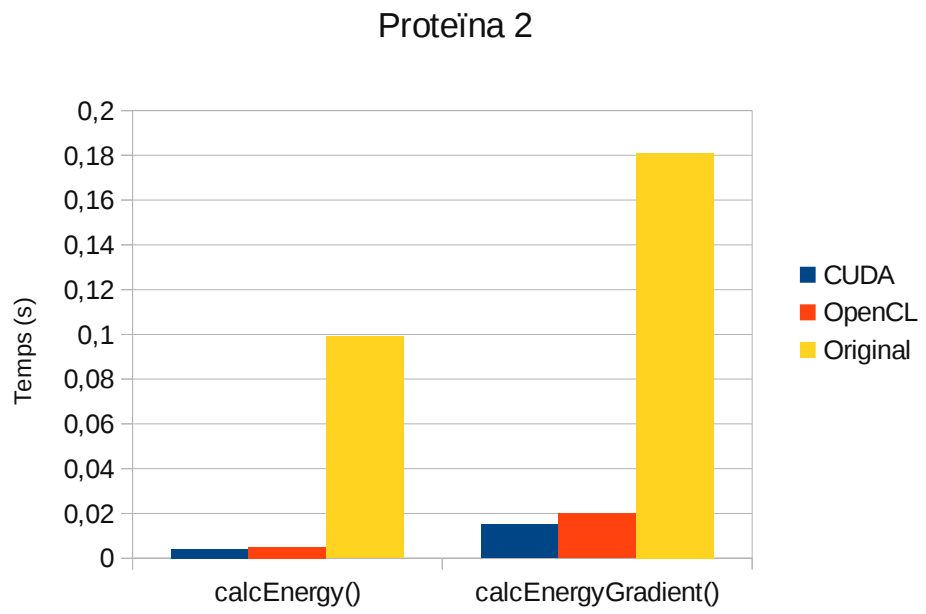


Proteïna 1	CUDA Temps	SpeedUp	OpenCL Temps	SpeedUp	Original Temps
calcEnergy	0,003	15	0,003	15	0,045
calcEnergyGradient	0,01	8,7	0,01	8,7	0,087

(Temps en segons)

Fig. 27 Resultats del temps d'execució i speed up respecte la versió seqüencial de les versions CUDA i OpenCL utilitzant la proteïna 1.

En la proteïna 1, el speed up assolit respecte la versió original es el mateix tant en CUDA com OpenCL, 15x en *calcEnergy* i 8,7x en *calcEnergyGradient* .



Proteïna 2	CUDA Temps	SpeedUp	OpenCL Temps	SpeedUp	Original Temps
calcEnergy	0,004	24,8	0,005	19,8	0,099
calcEnergyGradient	0,015	12,1	0,02	9,1	0,181

(Temps en segons)

Fig. 28 Resultats del temps d'execució i speed up respecte la versió seqüencial de les versions CUDA i OpenCL utilitzant la proteïna 2.

En la proteïna 2, amb més àtoms que la proteïna 1, la versio CUDA és una mica més ràpida que la versió d'OpenCL, amb un speed up de 24,8x i 12,1x per a *calcEnergy* i *calcEnergyGradient* respectivament, mentre que en OpenCL el speed up es de 19,8x i 9,1x

6.5 Observacions

S'ha observat que en situacions on el clúster Minotauro tenia molta demanda d'ús, les mesures del temps no eren correctes a no ser que reservéssim tot un node per realitzar les proves. No era suficient en reservar una CPU sencera i la seva corresponent targeta gràfica, era necessari reservar les dues CPUs del node per obtenir les mesures correctes. Una altra via per poder executar el test amb fiabilitat ha estat utilitzant un nou mode del Minotauro, que esta disponible des de el mes d'abril, on s'obre una sessió en un node en exclusiva per tal de poder realitzar execucions interactives.

7 Anàlisi de rendiment en la variació dels threads per bloc

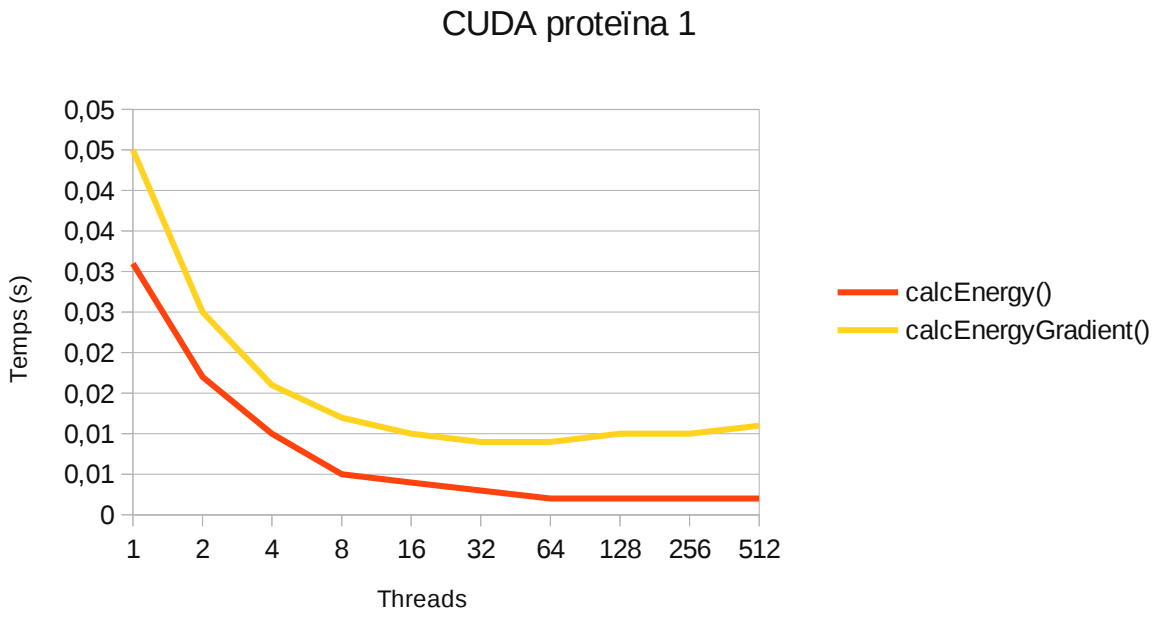
Al tenir finalitzada la implementació del codi a executar en la GPU, realitzarem un estudi per intentar millorar més el rendiment, ajustant els threads per bloc. Volem trobar la millor configuració i veure'n les diferencies entre la versió CUDA i OpenCL. En l'apartat 6.3.2.3, havíem experimentat amb el nombre de threads per bloc en l' algorisme d'ordenació Bitonic sort al ser exclusiu de la versió d'OpenCL.

Mesurarem els temps de les funcions *calcEnergy* i *calcEnergyGradient* , ja que la primera conté el kernel *kernelcalcEnergy* i la segona conté els kernels *kernelcalcEnergyGradient* i *kernelgradReduction2* .

Utilitzarem dues proteïnes, una de 14.132 àtoms i una altra de 31.313.

Les dades de temps, han estat obtingudes mesurant-les amb la macro *TIMER* i realitzant cinc execucions per obtenir-ne la mitjana.

Els resultats obtinguts en el clúster Minotauro, variant el nombre de threads entre 1 i 512, són els següents:

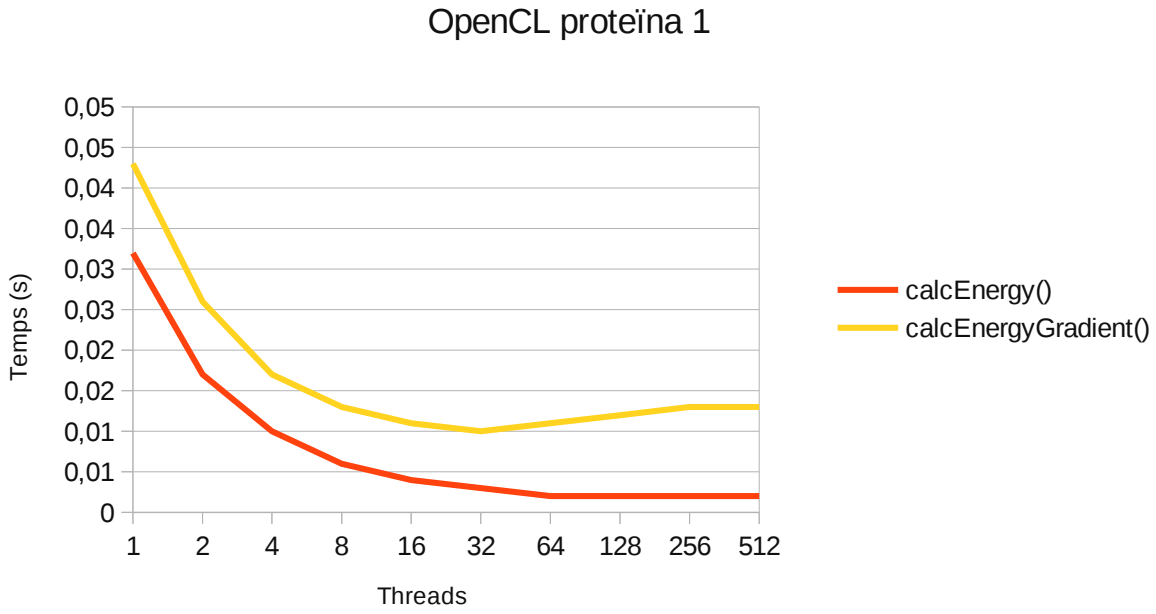


Proteïna 1 – CUDA	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
calcEnergy()	0,031	0,017	0,01	0,005	0,004	0,003	0,002	0,002	0,002	0,002
calcEnergyGradient()	0,045	0,025	0,016	0,012	0,01	0,009	0,009	0,01	0,01	0,011

(Temps en segons)

Fig. 29 Gràfic i taula amb els temps d'execució de la versió CUDA utilitzant la proteïna 1.

Al executar la versió CUDA per a la proteïna 1, el millor rendiment en la funció *calcEnergy* el trobem a partir de 64 threads, mentre que en *calcEnergyGradient* és en 32 i 64 threads.

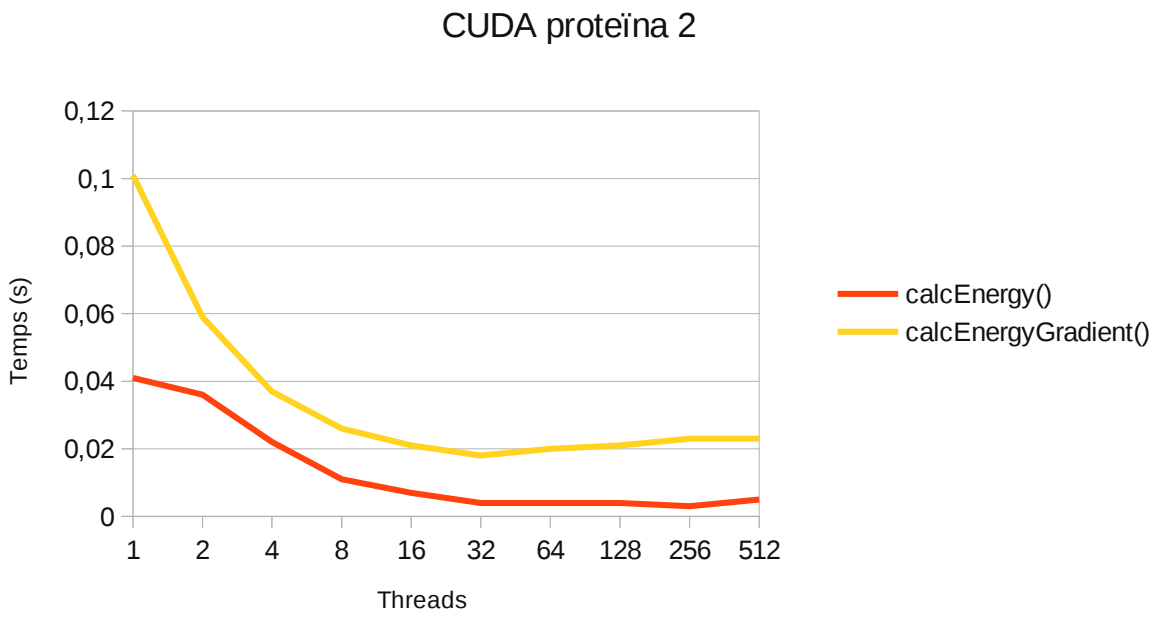


Proteïna 1 – OpenCL	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
calcEnergy()	0,032	0,017	0,01	0,006	0,004	0,003	0,002	0,002	0,002	0,002
calcEnergyGradient()	0,043	0,026	0,017	0,013	0,011	0,01	0,011	0,012	0,013	0,013

(Temps en segons)

Fig. 30 Gràfic i taula amb els temps d'execució de la versió OpenCL utilitzant la proteïna 1.

Al executar la versió OpenCL per a la proteïna 1, el millor rendiment en la funció *calcEnergy* el trobem a partir de 64 threads, mentre que en *calcEnergyGradient* és en 32 threads.

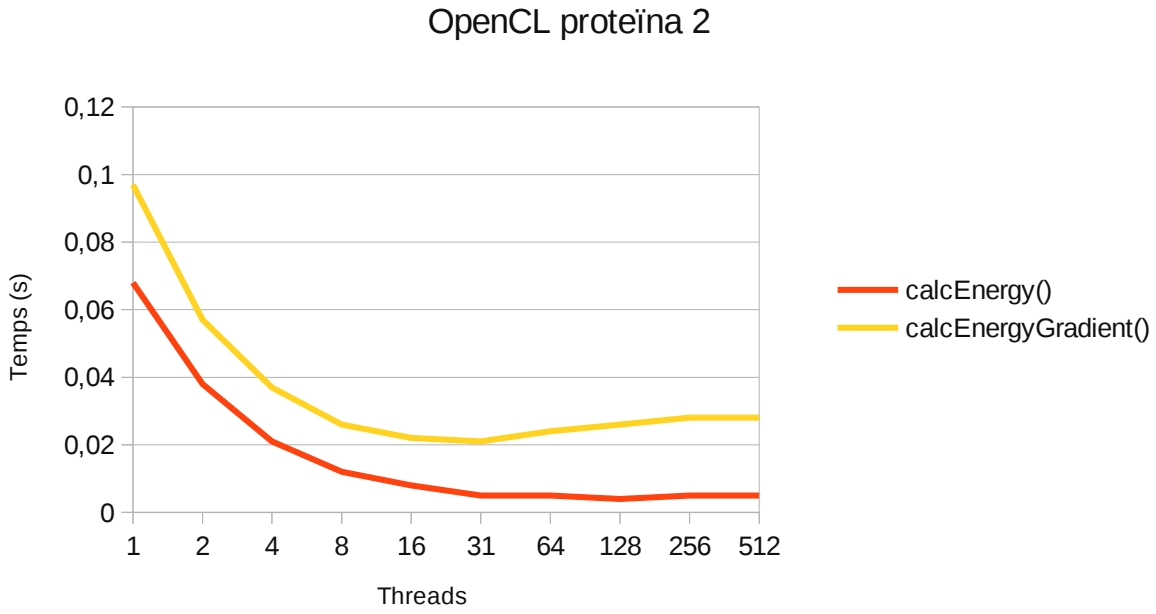


Proteïna 2 – CUDA	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
calcEnergy()	0,041	0,036	0,022	0,011	0,007	0,004	0,004	0,004	0,003	0,005
calcEnergyGradient()	0,101	0,059	0,037	0,026	0,021	0,018	0,02	0,021	0,023	0,023

(Temps en segons)

Fig. 31 Gràfic i taula amb els temps d'execució de la versió CUDA utilitzant la proteïna 2.

Al executar la versió CUDA per a la proteïna 2, el millor rendiment en la funció *calcEnergy* el trobem en 256 threads, mentre que en *calcEnergyGradient* és en 32 threads.



Proteïna 2 – OpenCL	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
calcEnergy()	0,068	0,038	0,021	0,012	0,008	0,005	0,005	0,004	0,005	0,005
calcEnergyGradient()	0,097	0,057	0,037	0,026	0,022	0,021	0,024	0,026	0,028	0,028

(Temps en segons)

Fig. 32 Gràfic i taula amb els temps d'execució de la versió OpenCL utilitzant la proteïna 2.

Al executar la versió OpenCL per a la proteïna 2, el millor rendiment en la funció *calcEnergy* el trobem a partir de 32 threads, mentre que en *calcEnergyGradient* és en 32 threads.

S'observa com el temps necessari per realitzar el càlcul de l'energia i el gradient en la proteïna 2 és aproximadament el doble que per la proteïna 1 (per a qualsevol nombre de threads), ja que té quasi el doble de INC.

Podem concloure que el comportament en CUDA i OpenCL en la variació del nombre de threads és similar. En tots els casos de *calcEnergyGradient* el millor resultat és utilitzant 32 threads per bloc.

En *calcEnergy* el millor resultat varia, però majoritàriament el trobem utilitzant 128 threads. Tot i així la diferencia entre 128 i 32 threads es mínima, només 0,001s.

Segons el comportament dels kernels (bloqueig de threads per llegir a memòria o accedir a una unitat de càlcul especial, ús de tots els blocs possibles, etc.) el nombre de threads optim varia i per tant es necessari comprovar que passa en cada cas.

En el càlcul de l'energia i el gradient ens trobem que el nombre de threads òptim és el mateix que el nombre de processadors que conté cada bloc, 32. Algunes de les causes podrien ser l'aprofitament màxim de tots els processadors possibles simultàniament i el poc bloqueig al que estan sotmesos els threads en execució.

En el cas del Bitonic sort de l'apartat 6.3.2 el millor rendiment era per a 128 threads, ja que ens trobem amb un codi que esta accedint constantment a la memòria global de la GPU, per tant els threads de cada bloc es bloquegen constantment i deixen pas a altres threads.

8 Integració de les noves funcions en PELE++

Ens interessa conèixer el speed up que tindria PELE++ si s'integressin les noves funcions del càlcul de l'energia i el gradient en la GPU. Fer la integració real de les funcions en PELE++ queda fora de l'abast del projecte degut a la seva complexitat.

Tot i que hem paral·lelitzat també *calcEnergy* al compartir part del codi de *calcEnergyGradient*, no l'inclourem al tenir poc pes en el profile i ens centrarem únicament en *calcEnergyGradient*.

Per veure la millora que obtindríem si integréssim les noves funcions, hem estudiat el rol que té la funció *calcEnergyGradient* dins el codi de PELE, que com hem apuntat en l'apartat 3.2.4., s'utilitza en l'etapa de la minimització. En cada iteració de PELE, es realitzen tres minimitzacions.

Dins la funció de minimització s'actualitzen les INC i les coordenades dels àtoms de la proteïna per trobar-ne la millor conformació. Per a cada configuració de INC, s'aniran modificant les coordenades dels àtoms de la proteïna, calculant-ne l'energia i el gradient per trobar les millors coordenades. Aquest algorisme s'implementa en el doble bucle del codi 20.

En el bucle extern, s'actualitzen les INC i les coordenades i tot seguit es realitza el càlcul de l'energia i el gradient. A continuació en el bucle intern s'anira modificant en cada iteració les coordenades dels àtoms i es mesurarà de nou l'energia i el gradient. El nombre d'iteracions del bucle extern (s'actualitzen les INC) i intern (nomes s'actualitzen les coordenades) està preestablert en les variables N i M respectivament. El rang de la variable N oscil·la entre 1 i 65 i el de la variable M entre 20 i 50.

```
...
copiar atoms GPU(); //createAtomVector()
mentre i<N fer
	...
	actualitzar INC();
	actualitzar coordenades();
	actualitzar INC GPU(); //createNBIVector() + createMapVectors()
	actualitzar coordenades GPU(); //copyCoordsToGPU()
	calcEnergyGradient();
	...
	mentre j<M fer
		...
		actualitzar coordenades();
		actualitzar coordenades GPU();
		calcEnergyGradient();
		...
		j++;

	fmentre

	i++;
fmentre
...
```

Codi 20 Bucle de la funció de la minimització afegint les instruccions de pre-procés de dades a la GPU en color verd.

Al conèixer l'ús de la funció *calcEnergyGradient* dins el bucle de l'etapa de minimització, podem situar les funcions de pre-procés de les dades dins el bucle (veure el codi anterior), i estipular el nombre de crides que es realitzen a *calcEnergyGradient* i a les funcions de pre-procés.

Com es fan tres minimitzacions, el temps total de les funcions relacionades amb el càlcul de l'energia i el gradient en una iteració de PELE és:

$$T = 3 \cdot (t_2 + N \cdot (t_1 + t_3 + t_4 + t_5 + M \cdot (t_1 + t_5)))$$

t_1 és el temps d'execució de la funció *copyCoordsToGPU*
t_2 és el temps d'execució de la funció *createAtomVector*
t_3 és el temps d'execució de la funció *createNBIVector*
t_4 és el temps d'execució de la funció *createMapVectors*
t_5 és el temps d'execució de la funció *calcEnergyGradient*

Fig. 33 Fórmula per calcular el temps d'execució total de les versions CUDA i OpenCL de la funció calcEnergyGradient en PELE++.

La fórmula de la versió seqüencial és més senzilla ja que no hem de preparar les dades:

$$T = 3 \cdot (N \cdot (t_5 + M \cdot (t_5))) = 3 \cdot N \cdot (M + 1) \cdot t_5$$

Fig. 34 Fórmula per calcular el temps d'execució total de la versió seqüencial de calcEnergyGradient en PELE++.

Utilitzarem els mateixos jocs de proves del profile per així poder utilitzar-ne les dades extretes, i calcular el speed up que assoliríem si integréssim el càlcul de l'energia i el gradient en PELE++. Els temps obtinguts per a cada versió utilitzant la macro *TIMER* (realitzant cinc execucions i traient-ne la mitjana), en les funcions que intervenen en el càlcul de l'energia i el gradient són els següents:

CUDA	**Proteïna 1**	**Proteïna 2**	**Proteïna 3**
copyCoordsToGPU()	0	0	0
createAtomVector()	0	0	0,001
createNBIVector()	0,003	0,042	0,094
createMapVectors()	0,003	0,010	0,018
calcEnergyGradient()	0,001	0,009	0,018

Taula 11 Temps d'execució en segons de les funcions CUDA per a cada un dels tres jocs de proves.

OpenCL	**Proteïna 1**	**Proteïna 2**	**Proteïna 3**
copyCoordsToGPU()	0	0	0
createAtomVector()	0	0.001	0,002
createNBIVector()	0,004	0,056	0,122
createMapVectors()	0,015	0,091	0,369
calcEnergyGradient()	0,001	0,01	0,021

Taula 12 Temps d'execució en segons de les funcions OpenCL per a cada un dels tres jocs de proves.

Seqüencial	**Proteïna 1**	**Proteïna 2**	**Proteïna 3**
calcEnergyGradient()	0,006	0,084	0,19

Taula 13 Temps d'execució en segons de la versió seqüencial de calcEnergyGradient per a cada un dels tres jocs de proves.

Com la funció *MapVector* no és igual en la versió CUDA i OpenCL degut a la funció d'ordenació

utilitzada, hem creat una hipotètica versió d'OpenCL utilitzant la mateixa funció d'ordenació Radix sort que s'utilitza en la versió CUDA, aplicant-li la diferencia de rendiment corresponent observada en la resta de les funcions CUDA i OpenCL. Així, la versió OpenCL* utilitza en tot moment els mateixos algorismes que la versió CUDA i ens permet comparar el rendiment de forma més justa.

OpenCL*	**Proteïna 1**	**Proteïna 2**	**Proteïna 3**
copyCoordsToGPU()	0	0	0
createAtomVector()	0	0.001	0,002
createNBIVector()	0,004	0,056	0,122
createMapVectors()	0,004	0,012	0,022
calcEnergyGradient()	0,001	0,01	0,021

Taula 14 Temps d'execució en segons de les funcions OpenCL per a cada un dels tres jocs de proves.*

Hem extret l'ús de la funció *calcEnergyGradient* i el speed up màxim teòric (utilitzant la llei d'Amdahl) que podríem obtenir al optimitzar-la, dels profiles obtinguts en cada un dels jocs de proves (veure Taula 15).

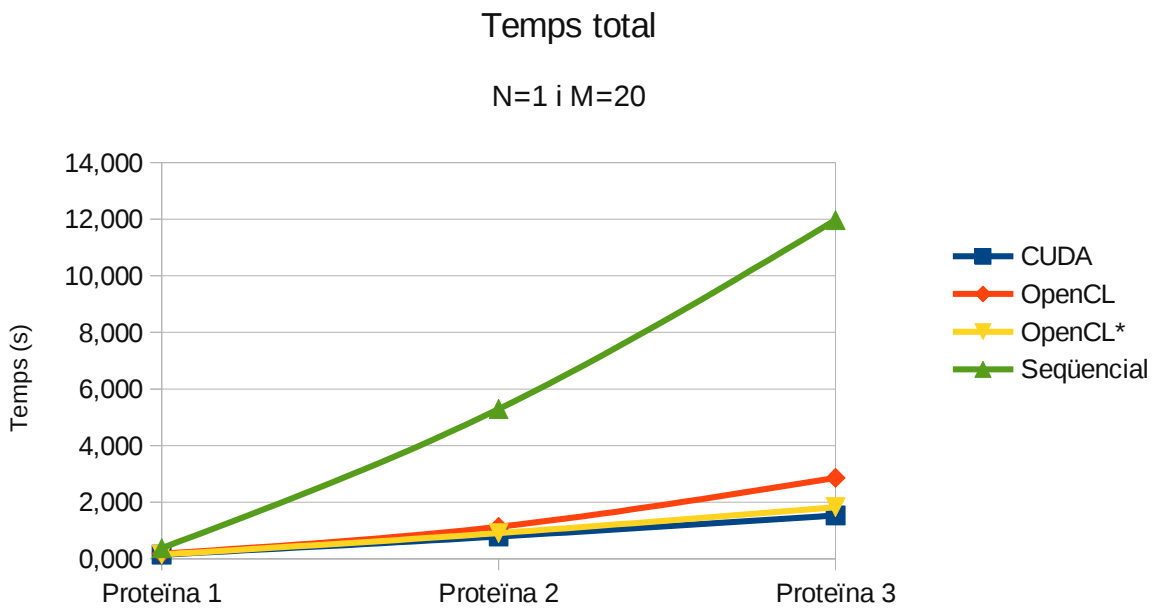
Profile	Proteïna 1	Proteïna 2	Proteïna 3
calcEnergyGradient	26,40%	32,00%	27,75%
Speed up màxim teoric	1,36	1,47	1,38

Taula 15 Percentatge d'ús i speed up màxim teòric de la funció calcEnergyGradient en cada un dels jocs de proves.

8.1 Resultats

Primer de tot, utilitzant els temps d'execució[12] de les funcions i la fórmula del temps total del càlcul de l'energia i el gradient dins el bucle de la minimització (figura 33 i 34), estudiarem el temps d'execució de les diferents implementacions (seqüencial, CUDA, OpenCL i OpenCL*) en el cas del mínim nombre de crides a *calcEnergyGradient* (N=1 i M=20) i en el cas màxim (N=65 i M=50). Ara veurem el temps total d'execució de *calcEnergyGradient* amb les funcions de pre-procés en les versions CUDA i OpenCL i el podrem comparar amb el temps de l'execució seqüencial. També hem calculat, partint del temps de cada versió, el speed up respecte la versió seqüencial.

12 En els casos on hem obtingut 0s, com la precisió de la mesura és de 0,001s, prendrem aquest valor enlloc de 0s per tal de situar-nos en el cas pitjor.



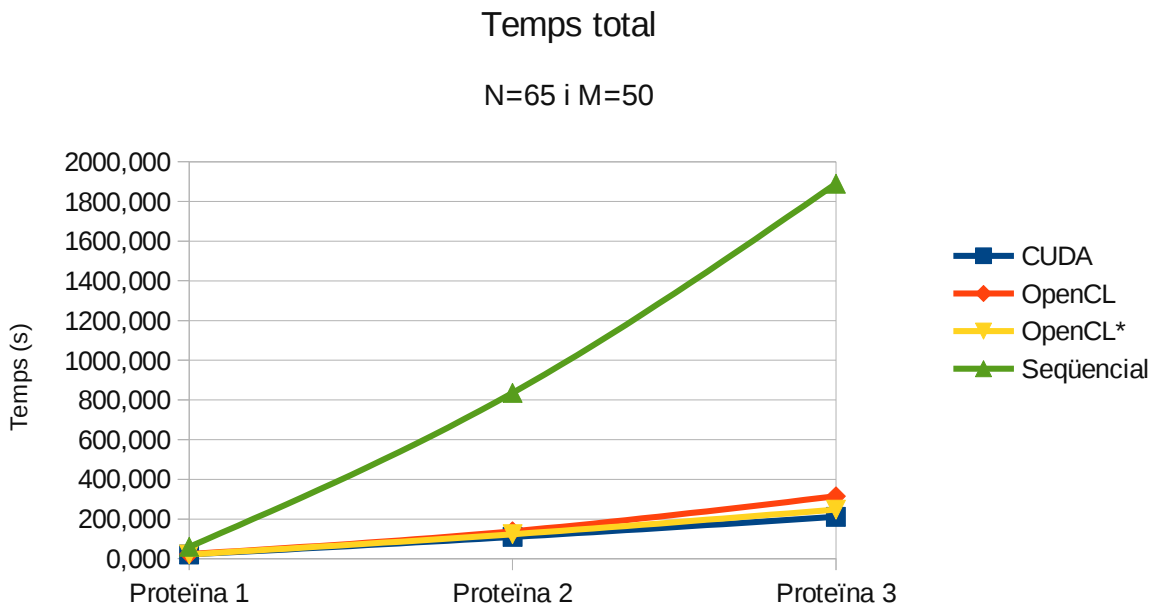
N=1 i M=20	Seqüencial			CUDA		
	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3
Temps total	0,378	5,292	11,970	0,145	0,787	1,534
SpeedUp	1,00	1,00	1,00	2,61	6,72	7,80

N=1 i M=20	OpenCL			OpenCL*		
	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3
Temps total	0,184	1,135	2,861	0,153	0,908	1,826
SpeedUp	2,05	4,66	4,18	2,48	5,83	6,56

(Temps total en segons)

Fig. 35 Gràfic i taula amb els resultats del temps i speed up en el cas del mínim nombre de crides a la funció calcEnergyGradient. La referència base per al càlcul del speed up es la versió seqüencial.

En el cas del mínim nombre de crides a *calcEnergyGradient* , podem observar en la figura 35 com obtenim una millora important de rendiment en les versions paral·lelitzades respecte la versió seqüencial, incrementant-se a mesura que creix la proteïna en les versions CUDA i OpenCL*. Això no passa en la versió OpenCL degut al fort increment de temps necessari en l'ordenació de les taules de mapeig de la proteïna 3.



N=65 i M=50	Seqüencial			CUDA		
	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3
Temps total	59,670	835,380	1889,550	21,061	109,591	210,796
SpeedUp	1,00	1,00	1,00	2,83	7,62	8,96

N=65 i M=50	OpenCL			OpenCL*		
	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3
Temps total	23,596	138,061	314,537	21,549	123,284	247,296
SpeedUp	2,53	6,05	6,01	2,77	6,78	7,64

(Temps total en segons)

Fig. 36 Gràfic i taula amb els resultats del temps i speed up en el cas del màxim nombre de crides a la funció calcEnergyGradient. La referència base per al càlcul del speed up es la versió seqüencial.

En el cas del màxim nombre de crides a *calcEnergyGradient* (resultats en la figura 36), el comportament és similar al del cas mínim, però obtenint un major speed up. Això es degut al augment del ràtio entre les funcions de pre-procés d'actualitzar les INC (*createNBIVector* i *createMapVectors*), amb la funció *calcEnergyGradient* . En el cas mínim per a cada actualització de les INC es realitzen 21 crides a calcEnergyGradient, mentre que en el cas màxim se'n realitzen 51. Aquest ràtio (actualització INC : *calcEnergyGradient*) ve determinat per la variable M i és $1:M+1$

Tant en el cas mínim com en el màxim, les versions CUDA i OpenCL*, a mesura que creix la proteïna augmenta el speed up. La versió OpenCL, al utilitzar la nostra funció d'ordenació es veu penalitzada i en la proteïna 3 el speed up es lleugerament inferior a la proteïna 2.

Ara que ja hem vist les diferencies de rendiment entre la versió seqüencial i les versions paral·lelitzades, calcularem el speed up real que aconseguiríem en PELE++ si s'implementessin les versions CUDA, OpenCL i OpenCL*, en el cas mínim i màxim de crides a *calcEnergyGradient* . Els resultats estan recollits en la taula 16.

Speed up real	CUDA			OpenCL			OpenCL*		
	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3
Cas mínim	1,19	1,37	1,32	1,16	1,34	1,27	1,19	1,36	1,31
Cas màxim	1,21	1,39	1,33	1,19	1,36	1,30	1,20	1,38	1,32

Taula 16 Speed up real de PELE++ si s'implementen les versions CUDA, OpenCL i OpenCL de calcEnergyGradient.*

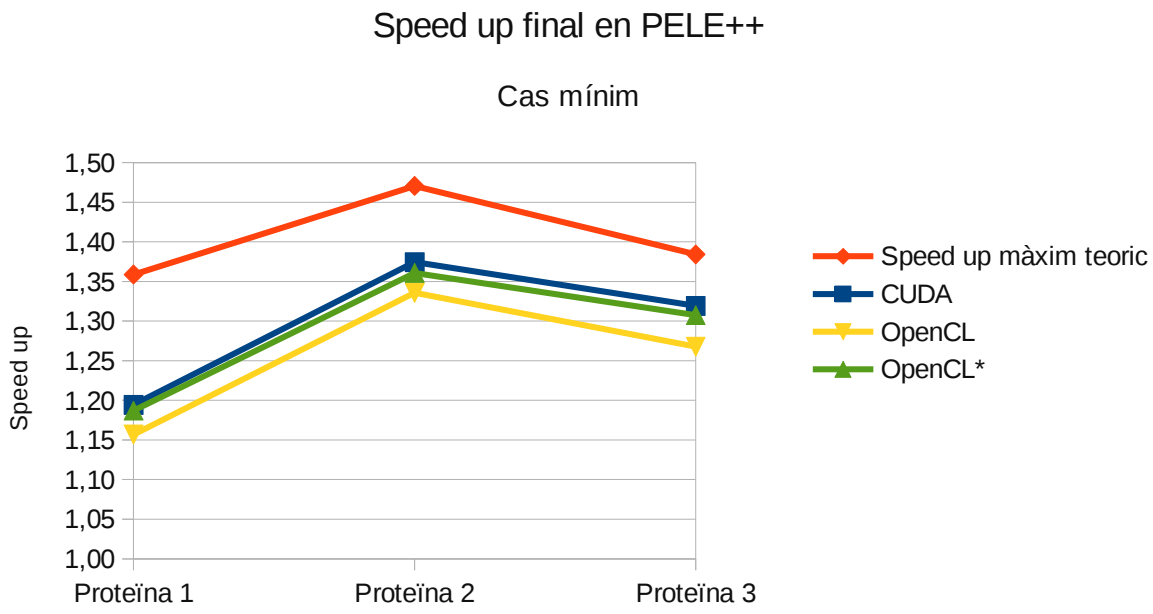


Fig. 37 Speed up en PELE++, en el cas mínim, de les versions CUDA, OpenCL i OpenCL respecte el màxim teòric.*

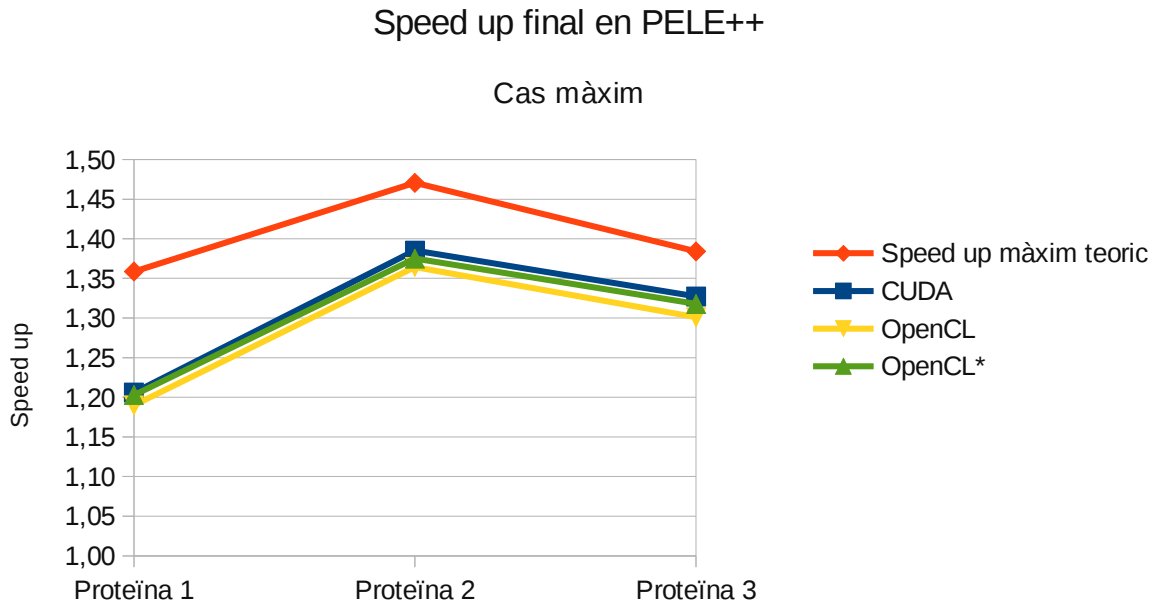


Fig. 38 Speed up en PELE++, en el cas màxim, de les versions CUDA, OpenCL i OpenCL respecte el màxim teòric.*

Com s'observa en els gràfics de les figures 37 i 38, a mesura que la proteïna creix ens aproximem al speed up màxim teòric que podríem aconseguir en PELE++ al optimitzar la funció *calcEnergyGradient* .

% de millora	CUDA			OpenCL			OpenCL*		
	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3	Proteïna 1	Proteïna 2	Proteïna 3
Cas mínim	16,27%	27,24%	24,19%	13,55%	25,14%	21,12%	15,75%	26,51%	23,52%
Cas màxim	17,08%	27,80%	24,65%	15,96%	26,71%	23,13%	16,87%	27,28%	24,12%

Taula 17 Percentatge de millora de PELE++ si s'implementen les versions CUDA, OpenCL i OpenCL de calcEnergyGradient en el cas de mínim i màxim nombre de crides a la funció.*

Si s'implementessin les noves versions en el càlcul de l'energia i el gradient a PELE++, la millora que suposaria la versió CUDA oscil·laria entre un 16,3 i 27,8%, la versió OpenCL entre un 13,5 i 26,7% i la versió OpenCL* entre un 15,7 i 27,3%.

9 Execució en una targeta gràfica AMD

Com tenim una versió en OpenCL que ens permet portar el programa a diferents sistemes, volem veure com varia el rendiment en una altra màquina. Per aquesta raó experimentarem amb una màquina totalment diferent que conté una CPU i GPU de la companyia AMD.

Les característiques tècniques de la CPU són les següents:

```
model name   : AMD A8-3850 APU with Radeon(tm) HD Graphics
cpu MHz      : 2900.000
cache size   : 1024 KB
cpu cores    : 4
bogomips     : 5800.30
```

La targeta gràfica és una AMD Radeon HD 6870 i té una arquitectura diferent a la Nvidia. Les característiques més destacades són les següents:

- 1GB de memòria
- 1120 stream processors (repartits en 14 grups)
- Ample de banda 134GB/s
- 2.016 Gflops (precisió simple)
- OpenCL 1.2

Aquesta GPU no té doble precisió en coma flotant, per tant haurem de crear una nova versió del codi canviant tots els tipus de dades double per float per tal de poder-lo executar. El nombre màxim de threads per bloc són 256 a diferencia de la Nvidia que eren 512.

Pel que fa al software de la màquina, el S.O. és Debian (kernel de linux 3.2.0-2-amd64) i el driver de OpenCL és de AMD.

Tornarem a utilitzar els mateixos jocs de proves que en l'apartat anterior.

9.1 Proves i resultats

Les dades de temps d'aquest apartat, han estat obtingudes mesurant-les amb la macro *TIMER* i realitzant cinc execucions per obtenir-ne la mitjana.

9.1.1 CPU

En la primera prova veurem quan triga l'execució seqüencial de *calcEnergyGradient* en la nova CPU utilitzant tres jocs de proves:

	Proteïna 1	Proteïna 2	Proteïna 3
Intel	0,006	0,084	0,19
AMD	0,006	0,084	0,184

Taula 18 Temps en segons, al executar la versió seqüencial de calcEnergyGradient en els processadors Intel i AMD.

Les dues CPUs tenen una potencia semblant, ja que la versió seqüencial del nostre codi tarda pràcticament el mateix.

9.1.2 Benchmark Bitonic sort

La segona prova és l'execució del benchmark creat per valorar el rendiment de l'algorisme Bitonic sort en OpenCL.

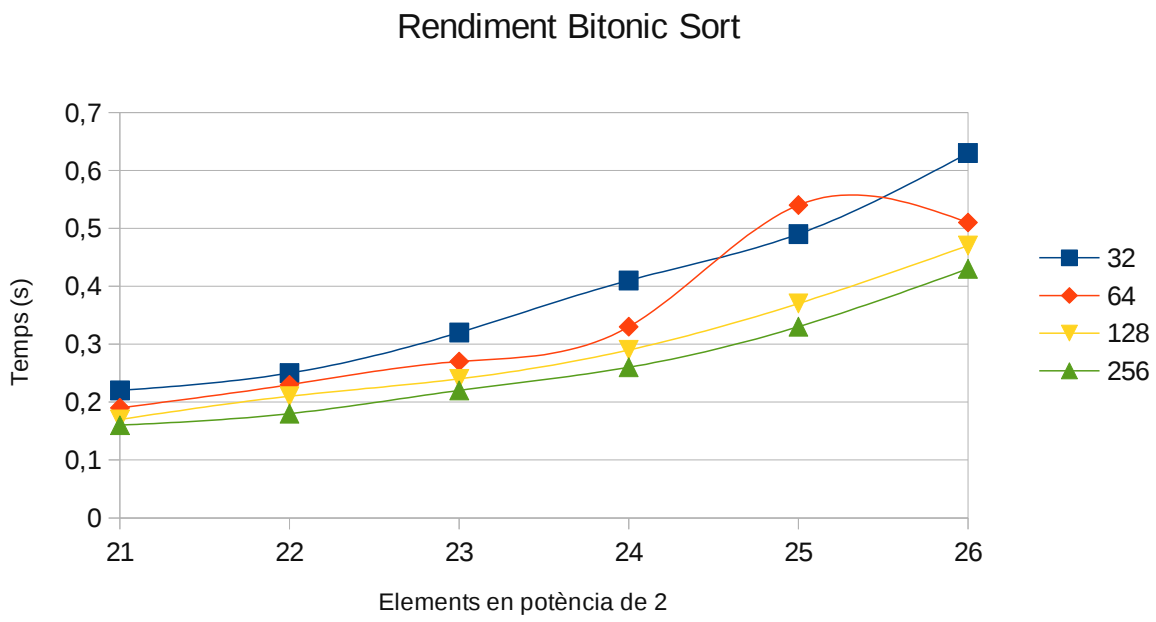


Fig. 39 Rendiment de l'algoritme Bitonic sort en la targeta AMD utilitzant de 32 a 256 threads per bloc.

El millor rendiment en la targeta AMD l'obtenim quan el nombre de threads és de 256. Com podem observar en la figura 39, el comportament quan tenim 64 threads sembla un artefacte, però en totes les execucions era igual.

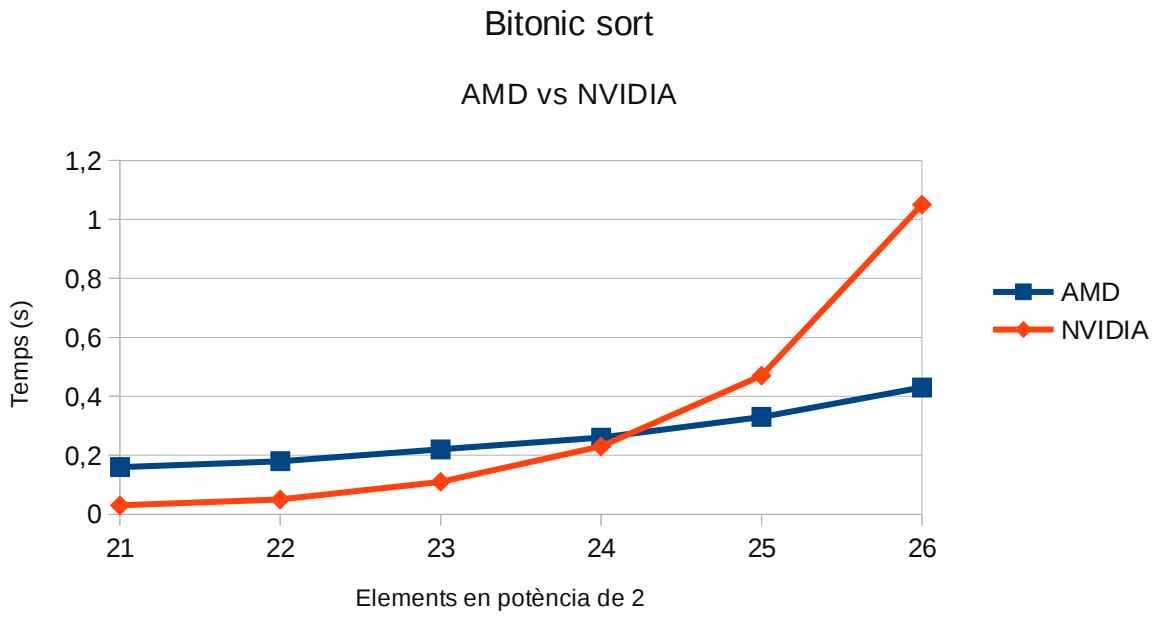


Fig. 40 Comparativa de rendiment en l'execució de l'algoritme Bitonic sort en una targeta AMD i Nvidia.

El comportament és diferent en les dues targetes gràfiques. En la targeta Nvidia, al doblar el nombre d'elements a ordenar es dobla el temps emprat, mentre que la targeta AMD no es dobla el temps quan doblem en número d'elements. Tot i que la targeta AMD tarda més quan tenim menys de 2^{24} elements, el creixement és més suau i al arribar a 2^{25} elements el rendiment ja es superior a la Nvidia. El rendiment obtingut al ordenar 2^{26} elements és més del doble, passant de 64 milions d'elements per segon en la targeta Nvidia a 156 milions d'elements per segon en la AMD.

Si el nombre de INC fos superior a 2^{24} elements, obtindríem un millor rendiment utilitzant la targeta AMD, però com en el nostre projecte la proteïna més gran té poc més de 2^{22} elements, trigarà més la ordenació en la targeta AMD que en la Nvidia.

9.1.3 Comportament en calcEnergy i calcEnergyGradient

Finalment provarem les nostres funcions variant el nombre de threads per bloc, amb la finalitat de veure el comportament de la targeta AMD i els temps que tarden en executar-se les funcions. Ens hem trobat que nomes hem pogut provar-ho fins a 64 threads per bloc ja que amb 128 el programa es queda bloquejat i s'ha de reiniciar el sistema. Això es degut al diferent funcionament de la targeta AMD, ja que el nombre de threads màxim dependrà dels registres necessaris pel kernel. En aquest cas es menys que el màxim teòric de 256 [10].

Com hem canviat el codi i els càlculs són tots a precisió simple, s'han pres totes les mesures variant el nombre de threads en el sistema AMD i Nvidia. Recordem que les funcions que executen kernels i per tant es veuran afectades pel nombre de threads són *createMapVectors* , *calcEnergy* i *calcEnergyGradient* .

AMD	Proteïna 11			Proteïna 12			Proteïna 13		
	16	32	64	16	32	64	16	32	64
copyCoordsToGPU()	0	0	0	0	0	0	0,001	0,001	0,001
createAtomVector()	0	0	0	0,001	0,001	0,001	0,001	0,001	0,001
createNBIVector()	0,003	0,003	0,003	0,061	0,062	0,062	0,136	0,136	0,135
createMapVectors()	0,208	0,206	0,02	0,505	0,502	0,505	1,189	1,165	0,171
calcEnergy()	0,005	0,004	0,004	0,01	0,009	0,007	0,013	0,01	0,01
calcEnergyGradient()	0,009	0,008	0,008	0,024	0,022	0,022	0,028	0,023	0,025

(Temps en segons)

Taula 19 Temps d'execució de les funcions en la màquina AMD, variant-ne el nombre de threads (16, 32, 64) i utilitzant tres jocs de proves.

Nvidia	Proteïna 11			Proteïna 12			Proteïna 13		
	16	32	64	16	32	64	16	32	64
copyCoordsToGPU()	0	0	0	0	0	0	0	0	0
createAtomVector()	0	0	0	0,001	0,001	0,001	0,001	0,001	0,001
createNBIVector()	0,003	0,003	0,003	0,04	0,042	0,041	0,091	0,091	0,09
createMapVectors()	0,014	0,014	0,014	0,083	0,081	0,081	0,35	0,347	0,344
calcEnergy()	0,001	0,001	0,001	0,003	0,003	0,002	0,005	0,004	0,003
calcEnergyGradient()	0,001	0,001	0,001	0,007	0,008	0,009	0,015	0,017	0,019

(Temps en segons)

Taula 20 Temps d'execució de les funcions en el clúster Minotauro, variant-ne el nombre de threads (16, 32, 64) i utilitzant tres jocs de proves.

En el sistema AMD, tot i que els valors són molt semblants en totes les configuracions, el millor rendiment s'obté utilitzant 32 threads per bloc. En el cas de Nvidia en aquesta versió amb simple precisió el millor rendiment s'obté utilitzant 16 threads per bloc, enlloc dels 32 threads de la versió amb doble precisió.

Pel que fa al temps d'execució de les funcions, en totes les funcions la targeta Nvidia és més ràpida.

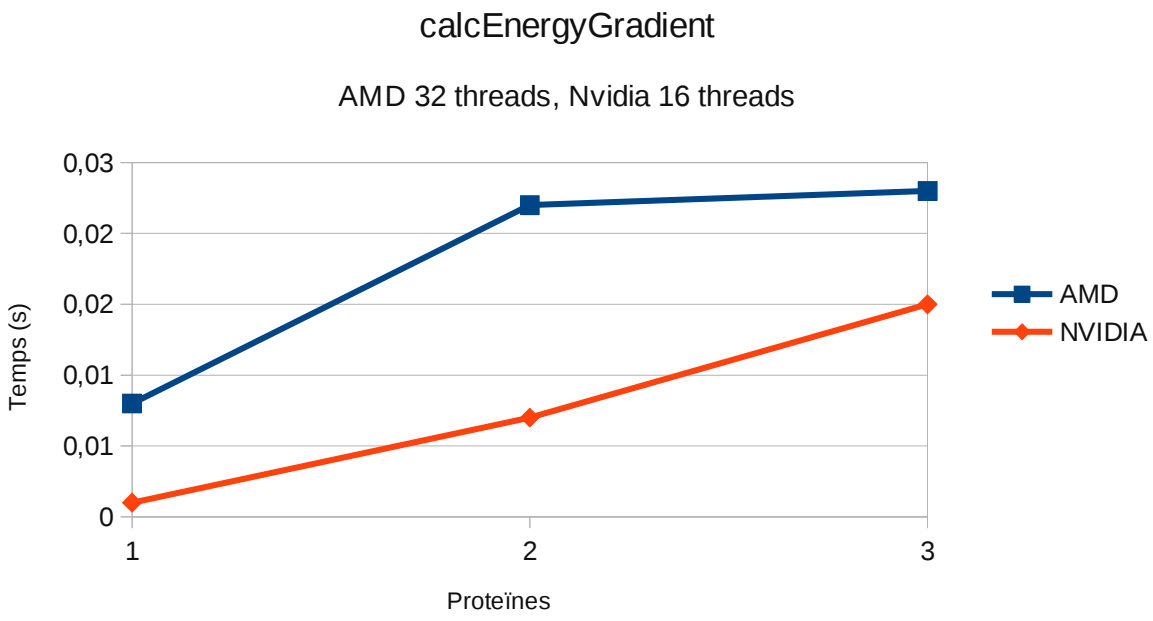


Fig. 41 Comparativa de rendiment entre la targeta AMD i Nvidia en la funció calcEnergyGradient.

Centrant-nos en la funció *calcEnergyGradient* , la targeta Nvidia obté un millor rendiment que la targeta AMD. El comportament en funció de la mida de la proteïna és molt diferent entre arquitectures tal i com es pot apreciar en el gràfic de la figura 41. La targeta Nvidia té un creixement constant i bastant lineal a mesura que creix la proteïna, mentre que la AMD en canvi, entre la primera proteïna de 1.231 àtoms i la segona de 14.132 l'increment de temps és de 14s, mentre que entre la segona i la tercera de 31.313 àtoms (on es dobla el nombre d'INC a calcular) és només de 1s. Necessitaríem més proteïnes i de mida més gran per comprendre millor el rendiment de la targeta AMD.

10 Ús d'OmpSs en el càlcul del gradient

OmpSs és un model de programació que exten OpenMP per tal de suportar paral·lelisme de tasques asíncron i sistemes heterogenis [11]. Les noves directives introduïdes en OmpSs per poder controlar les dependències de dades en el codi, permeten crear un graf de dependències per tal d'ordenar la seqüencia de tasques en la que s'ha partit el codi el codi inicial, i que el runtime d'OmpSs les executi en l'ordre correcte per garantir la correctesa de les dades. OmpSs també permet la integració de kernels CUDA per tal que el runtime s'encarregui d'executar-los en varies GPUs. La finalitat de OmpSs és facilitar la paral·lelització al desenvolupador.

El nostre objectiu es veure com podem realitzar usant OmpSs la paral·lelització del càlcul del gradient en la CPU. Per implementar-ho, hem afegit a la versió seqüencial de la funció *calcEnergyGradient* un pragma de OmpSs. Aquest pragma es situa en el bucle que realitza el càlcul de l'energia i gradient de totes les INC, indican les variables a protegir en el bucle per tal que OmpSs generi un graf de dependències. El runtime d'OmpSs, crearà per a cada iteració del bucle una tasca i programarà la seva execució, segons el graf de dependències, en tants processadors com li indiquem.

El pragma que hem introduït (codi 21), protegeix les posicions del vector gradient (per tal d'evitar la "situació de competició" que ja hem descrit en l'apartat 6.1.1) indicant que les tres posicions de l'àtom *a* i *b* en el vector són d'entrada i sortida. També hem indicat que el vector que conté les INC es compartit per a totes les tasques i la resta de variables que tenim dins el bucle són privades de cada tasca.

```
#pragma omp task inout(grad[a->ix;3], grad[b->ix;3]) shared(nonBondingInteractions) \
private(dx,dy,dz,dist2,cgij,disti,dist2i,sig2,epsij,rr2,rr6,rr12,electrostatic,lennard_j
ones, dfeldr, dfljdr, dfdr, drdx, drdy, drdz)
```

Codi 21 Pragma introduït en el bucle de la funció calcEnergyGradient.

Al finalitzar el bucle hem d'afegir un altre pragma per esperar-nos fins que finalitzin totes les tasques.

```
#pragma omp taskwait
```

Codi 22 Pragma per esperar la finalització de les tasques.

Com es pot veure les modificacions en el codi són mínimes i ha sigut molt fàcil d'implementar.

Varem utilitzar dos jocs de proves i els resultats obtinguts van ser un correcte càlcul del gradient, però el temps d'execució (obtingut utilitzant la macro *TIMER*) va ser molt superior a la versió seqüencial.

Versió	**Proteïna 1**	**Proteïna 2**
Original	0,006	0,084
OmpSs	0,43	34,2
OmpSs + unroll	0,32	26,4

Taula 21 Temps en segons d'executar el càlcul del gradient.

Es van utilitzar diferent nombre de processadors per veure'n l'escalabilitat i el millor resultats va ser per 1 processador. Això vol dir que les tasques no es poden executar en paral·lel i l'increment de temps al utilitzar més processadors es degut al overhead d'enviar les tasques a diversos

processadors.

La millor configuració del runtime de OmpSs va ser la següent:

```
NX_PES=1 NX_GPUS=0 NX_ARGS="--throttle numtasks --throttle-limit 10 --schedule wf --wf-local-policy FIFO
```

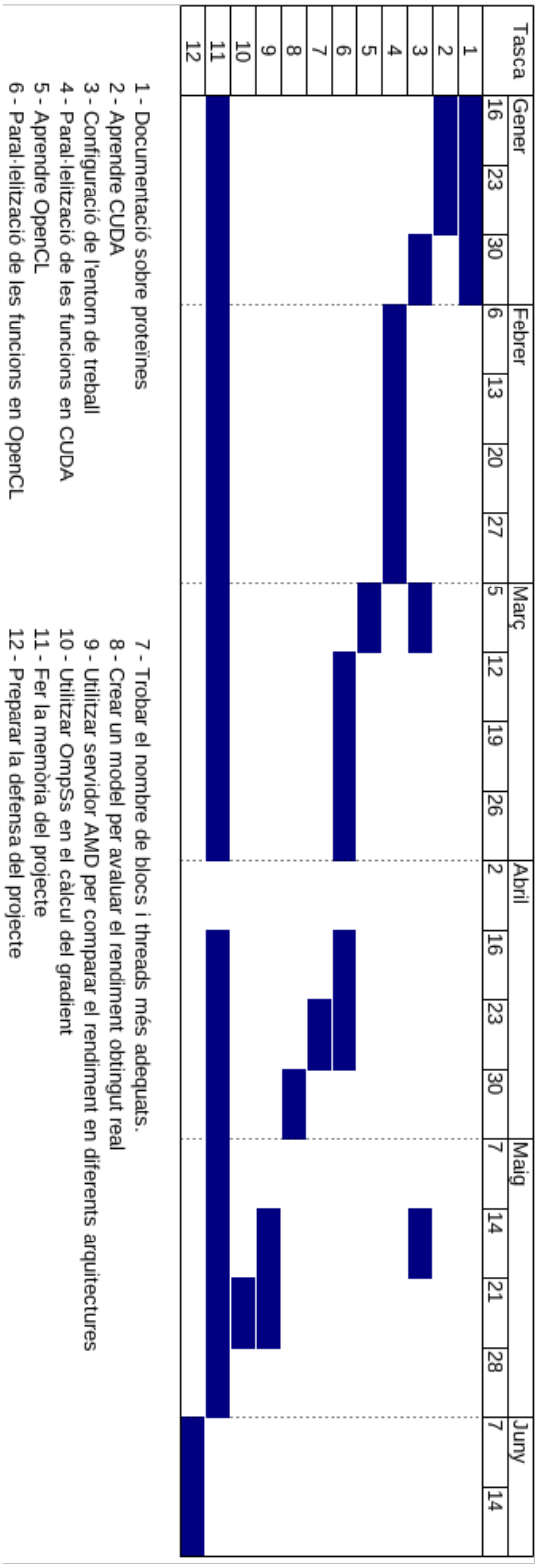
Per entendre l'increment de temps en el càlcul del gradient en la versió OmpSs, vam extreure unes traces amb el programa *extrae* . Les traces les vam obrir amb el programa *paraver* i vam veure com la gran part del temps d'execució era del runtime d'OmpSs i el temps de les tasques era extremadament petit. Això es degut a la baixa granularitat del càlcul de l'energia i el gradient, ja que el càlcul es molt ràpid.

Per augmentar-ne la granularitat vam aplicar un loop unrolling de grau 2, però els resultats no van millorar massa tal i com es pot observar en la taula 21. La millora de temps es pot atribuir al fet de crear la meitat de tasques i per tan reduir-ne el seu temps de creació.

Com a conclusió final, s'ha vist que OmpSs no es el millor paradigma per resoldre el càlcul del gradient en paral·lel, degut a la forta dependència de dades en l'actualització del vector del gradient i per tant no es possible accedir-hi concurrentment. També s'ha vist com els càlculs de l'energia i el gradient són molt petits i per tant la granularitat es massa fina com per compensar l'overhead de crear les tasques i repartir-les en varis processadors. Tot i així OmpSs ens ha sorprès per la seva facilitat d'ús. Ens ha permès fer una paral·lelització (encara que improductiva) que a priori era molt difícil afegint només 2 línies. Per aquesta raó serà un bon candidat a provar per la paral·lelització d'altres parts del codi de PELE++.

11 Planificació i costos

El projecte ha tingut una durada de 6 mesos. La planificació del projecte ha estat la següent:



Els costos d'aquest projecte són els següents:

- Un cap de projecte que dona suport al programador. Treballarà en el projecte una hora al dia i cobrarà 20€/h.
- El programador encarregat de realitzar la paral·lelització del codi. Treballarà 8 hores al dia i el cost és de 1700€ mensuals.
- El cost del material d'oficina utilitzat durant tot el projecte (paper, toner, boligrafs,...) és de 50€.
- Ordinador portàtil del programador 600€.
- La superfície d'oficina[13] ocupada té el següent cost: 12,5€/m^2*5m^2=62,5€ mes
- Consum elèctric de l'ordinador portàtil (40W)
- Electricitat extra gastada del supercomputador al dia de mitja:
  - El consum de la CPU quan està executant-se al 100% es de 80W. En el nostre projecte s'ha estimat que les proves d'execució del nostre codi són de 10min i l'ús de l'entorn de programació eclipse es de 24min al dia al 100%.
  - El consum de la GPU quan esta al 100% és de 250W. Les nostres proves funcionen acumulen 1 min al 100%.
- Preu KWh: 0,15€

	Cost unitari	**Unitats**	**Cost final**
Cap de projecte	440,00 €	5	2.200,00 €
Programador	1.700,00 €	5	8.500,00 €
Material d'oficina	50,00 €	1	50,00 €
Ordinador de treball	600,00 €	1	600,00 €
Oficina	62,50 €	5	312,50 €
Electricitat ordinador	1,06 €	5	5,28 €
Electricitat supercomputador	13,70 €	5	68,50 €
Cost total del projecte			**11.736,28 €**

Taula 22 Taula amb tots els costos del projecte i el cost total.

13 Preu del metre quadrat obtingut de [12].

12 Conclusions finals i futures millores

S'ha assolit l'objectiu d'executar i millorar el rendiment del càlcul de l'energia i el gradient de les interaccions no covalents utilitzant la GPU.

Quan es treballa en la GPU, la preparació de les dades i la copia a la memòria de la targeta gràfica porta molt de temps, per tant, tot i que els càlculs són molt ràpids, si el volum d'aquests càlculs no són molt importants no guanyarem temps.

A l'hora d'elegir un framework de programació, hem vist com les funcions CUDA són una mica més ràpides que les seves homònimes en OpenCL en el mateix hardware. Aquest fet unit a la mancança en OpenCL d'una biblioteca oficial amb funcions comuns d'alt rendiment (com la biblioteca Thrust de CUDA), fa que ens decantem en l'ús de CUDA.

En l'execució del test en la targeta Nvidia, el rendiment és constant, és a dir, si es doblen els elements a computar necessita aproximadament el doble de temps. Aquest comportament s'ha pogut observat tant en el càlcul de l'energia i el gradient com en l'algorisme Bitonic sort. En canvi la targeta AMD té un comportament totalment diferent: quan més gran es l'entrada, major es el rendiment obtingut.

El fet d'executar el mateix programa en dues targetes diferents, ens ha mostrat l'elevada dependència del hardware i com varia el rendiment entre les diferents arquitectures de GPUs. Es necessita conèixer el hardware per poder explotar-ne les seves característiques i optimitzar de forma adequada el codi. Això obliga al desenvolupador a crear optimitzacions especifiques a cada dispositiu.

Si es vol utilitzar OmpSs en el càlcul del gradient s'hauria de buscar una estratègia i adaptar les dades, com s'ha fet en la versió CUDA amb les taules de mapeig, per tal de poder realitzar el càlcul del gradient en paral·lel.

De cara el futur, per millorar el rendiment de PELE++, s'hauria d'optimitzar el càlcul del solvent i el càlcul de les energies de les interaccions covalent, ja que respecte el càlcul de l'energia i el gradient de les INC (tal i com s'ha pogut observar en els gràfics de la figura 37 i 38 de l'apartat 8) el marge de millora és petit. Tot i així, les millores que es podrien fer en el càlcul de l'energia i el gradient utilitzant la GPU són:

- Adaptar el codi per utilitzar més d'una targeta gràfica en el bucle de la minimització, ja que el bucle intern podria repartir les iteracions del càlcul de l'energia i el gradient entre varies GPUs.
- Optimitzar a baix nivell els kernels del càlcul de l'energia i el gradient, intentant explotar al màxim les característiques de la GPU on es vagi a executar. Per exemple, utilitzant funcions matemàtiques i operacions (com la FMA[14]) específiques de CUDA o intentar explotar de forma més eficient la jerarquia de memòria de la targeta gràfica.
- Trobar un algorisme d'ordenació per a OpenCL igual de ràpid que la funció sort de la biblioteca Thrust de CUDA.
- En el model hem treballat sempre en el cas pitjor i per tant la preparació de les dades esta just abans de la funció *calcEnergyGradient* . Aprofitant l'asincronisme en l'ús de la GPU, al implementar en PELE++ el càlcul de l'energia i el gradient en la GPU, es podria intentar situar les funcions *copyCoordsToGPU* , *createNBIVector* i *createMapVectors* del bucle de la funció de minimització, de tal manera que treballes la CPU i la GPU de forma simultània.

14 És una unitat hardware que realitza la següent operació: $a \leftarrow a + (b \cdot c)$

13 Bibliografia i referències

13.1 Bibliografia

Llibres:

Thomas M. Devlin, *Bioquímica*, 4ª ed. Reverté.

Carlos Gómez-Moreno Calera i Javier Sancho Sanz, *Estructura de proteínas*, 1ª ed. Ariel.

Jason Sanders i Edward Kandrot, CUDA by Example: An Introduction to Gneeral-Purpose GPU Programming, 1ª ed. Addison-Wesley.

Munhsi, Gaster, Mattson, Fung i Ginsburg, *OpenCL Programming Guide*, 1ª ed. Addison-Wesley.

Guies:

Nvidia CUDA C Programming Guide, versió 4.0.

Webs:

www.wikipedia.org

www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/

www.nvidia.com

13.2 Referències

[1] Thomas M. Devlin, *Bioquímica*, 4ª ed. Reverté.
[2] Kenneth W. Borrelli, Andreas Vitalis, Raul Alcantara i Victor Guallar, «Protein Energy Landscape Exploration. A novel Monte Carlo Based Technique».
[3] Carlos Gómez-Moreno Calera i Javier Sancho Sanz, *Estructura de proteínas*, 1ª ed. Ariel.
[4] Jay W. Ponder i David A. Case, «Force fields for protein simulations».
[5] Hun Joo Myung, Kwang Jin Oh, Ryuji Sakamaki, Tetsu Narumi, Kenji Yasuoka i Sik Lee, «Accelerating Molecular Dynamics SImulations Using Graphics Processing Unit».
[6] «Minotauro». [Online]. Available: http://www.bsc.es/sites/default/files/public/about/news/minotauro-29012012-computing.pdf. [Accessed: 29-abr-2012].
[7] «thrust: Sorting». [Online]. Available: http://docs.thrust.googlecode.com/hg/group__sorting.html. [Accessed: 02-mar-2012].
[8] «Bitonic sort». [Online]. Available: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm. [Accessed: 28-abr-2012].
[9] «RadixSorting - back40computing - High performance GPU radix sorting in CUDA - Fast and efficient software primitives for GPU computing - Google Project Hosting». [Online]. Available: http://code.google.com/p/back40computing/wiki/RadixSorting. [Accessed: 29-abr-2012].
[10] «OpenCL forum: OpenCL local work group size». [Online]. Available: http://www.macresearch.org/opencl-local-work-group-size#comment-16465. [Accessed: 12-jun-2012].
[11] «OmpSs | Programming Models @ BSC». [Online]. Available: http://pm.bsc.es/ompss. [Accessed: 25-may-2012].

[12] «El precio del alquiler en Barcelona capital se estabiliza en noviembre». [Online]. Available: http://www.elplural.com/2011/12/14/el-precio-del-alquiler-en-barcelona-capital-se-estabiliza-en-noviembre/. [Accessed: 30-may-2012].
[13] Kayvon Fatahalian i Mike Houston, «A Closer Look at GPUs».
[14] Nvidia, «Nvidia CUDA C programming guide». .
[15] Ginés D. Guerrero, Horació Pérez-Sanchez, Wolfgang Wenzel, José M. Cecilia i José M-García, «Effective Parallelization of Non-Bonded Interactions Kernel for Virtual Screening on GPUs».
[16] James Phillips i John Stone, «GPU Acceleration of Molecular Modeling Applications», Beckman Institute, UIUC.

14 Glosari

Ample de banda: Quantitat d'informació que es pot enviar en un canal de dades en un periode de temps determinat. S'acostuma a donar en bits/segon.

Blade: Targeta que conte com a mínim un microprocessador, memòria i busos de conexió, i s'utilitzen per montar servidors.

Conformació: Forma i disposició dels element que pren una estructura.

CPU: Central Processing Unit, processador principal d'un ordinador.

DM: Dinàmica molecular

Framework: Conjunt d'eines dissenyades per facilitar el desenvolupament software, sobre una plataforma coneguda.

GPU: Graphics Processing Unit, és el processador de les targetes gràfiques.

INC: Interacció no covalent.

Loop unrolling: Tecnica d'optimització basada en desplegar els bucles.

Overhead: Temps addicional degut a un increment d'instruccions per tal de realitzar alguna tasca.

Outlayer: Valor que dista molt de la mitjana.

Padding: Farciment per omplir de dades alguna estructura.

Pipeline: Etapes d'un processador en les que es realitzen operacions.

Profile: Tècnica d'extreure informació relativa al temps d'execució d'un programa.

Runtime d'OmpSs: Funcionalitats d'OmpSs que s'executen juntament amb el programa que fa ús del sistema OmpSs.

Speed up: Factor de millora entre dues execucions o codis. Es calcula dividint el temps inicial entre el temps final.

Thread: Fil d'execució d'instruccions en un processador.

Throughput: Volum de treball realitzat.

I Annex

I.I Jocs de proves

Cada joc de proves utilitzat en el projecte contenia una de les següents proteïnes:

Nom	Minimitzada	Nombre d'àtoms	Nombre de parelles
MDL1	NO	2.515	1.912.093
3UTG	NO	31.313	4.174.131
1AAR	SI	1.231	138.135
3QLQ	SI	14.132	1.926.008
3UTG	SI	31.313	4.239.271

En el transcurs del projecte vam veure com la proteïna 3UTG no minimitzada, al executar el test, donava un petit error de precisió en el càlcul del gradient de dos àtoms. La causa és el rang màxim de representació del tipus de dades double, ja que el gradient d'aquests dos àtoms va créixer molt, i al representar la part entera no es podia representar el vuitè decimal que comprova el test. Com el fitxer del joc de proves que conté el resultat del gradient de tots els àtoms no conté aquest vuitè decimal, el test dona error de precisió.

I.II Codis

I.II.I CUDA iteracio1

Codi de *calcEnergyGradient* on creem el vector de *NewNBP* , el copiem a memòria, executem el kernel, recollim els resultats i en fem la reducció, primer de l'energia i després del gradient:

```
double CUDANonBondingTermVacuumCalculator::calcEnergyGradient(double * coords, double *
grad, vector< NonBondPair > & nonBondingInteractions)
{
	double *dev_result, *result, res=0;
	int numPairs = nonBondingInteractions.size();
	int num_blocks = ceil((double)numPairs  / (double) THREADS_PER_BLOCK);
	int *dev_numPairs, i;
	NewNBP *v, *dev_v;
	ThreeCoords *dev_coords, *tcoords;

	//Reserva memoria pel vector de parelles i els vectors on guardar els resultats
	v=(NewNBP*)malloc(numPairs*sizeof(NewNBP));
	tcoords=(ThreeCoords*)malloc(numPairs*sizeof(ThreeCoords));
	result=(double*)malloc(num_blocks*sizeof(double));

	//Creació del vector de parelles
	createCudaVector(coords, nonBondingInteractions, v);

	//Reservem memoria a la GPU
	cudaMalloc( (void**)&dev_numPairs, sizeof(int));
	cudaMalloc( (void**)&dev_v, numPairs*sizeof(NewNBP) );
	cudaMalloc( (void**)&dev_coords, numPairs*sizeof(ThreeCoords) );
	cudaMalloc( (void**)&dev_result, num_blocks*sizeof(double));

	//Copia de la memoria de la CPU a la GPU
	cudaMemcpy( dev_numPairs, &numPairs, sizeof(int), cudaMemcpyHostToDevice);
	cudaMemcpy( dev_v, v, numPairs*sizeof(NewNBP), cudaMemcpyHostToDevice);

	//Executem el Kernel
	kernelcalcEnergyGradient<<<num_blocks,THREADS_PER_BLOCK>>>(dev_v, dev_numPairs,
```

```
dev_result, dev_coords);

	//Copiem el resultats de la memoria de la GPU a la CPU
	cudaMemcpy(result, dev_result, num_blocks*sizeof(double), cudaMemcpyDeviceToHost);
	cudaMemcpy(tcoords, dev_coords, numPairs*sizeof(ThreeCoords),
cudaMemcpyDeviceToHost);

	//Fem les reduccions primer del resultat i a continuació del gradient
	for(i=0;i<num_blocks;i++){
		res+=result[i];
	}
	for(i=0; i<numPairs;i++){
		grad[ nonBondingInteractions[i].first->ix ] += tcoords[i].x;
		grad[ nonBondingInteractions[i].first->iy ] += tcoords[i].y;
		grad[ nonBondingInteractions[i].first->iz ] += tcoords[i].z;
		grad[ nonBondingInteractions[i].second->ix ] -= tcoords[i].x;
		grad[ nonBondingInteractions[i].second->iy ] -= tcoords[i].y;
		grad[ nonBondingInteractions[i].second->iz ] -= tcoords[i].z;
	}

	//Alliberem la memoria solicitada tant a la GPU com CPU
	cudaFree(dev_result);
	cudaFree(dev_v);
	cudaFree(dev_numPairs);
	cudaFree(dev_coords);
	free(result);
	free(tcoords);
	free(v);

	return res;
}
```

Codi parcial de *kernelcalcEnergyGradient* mostrant com es desa el gradient en el vector de *ThreeCoords* i es fa la reducció dels threads del bloc:

```
__global__ void kernelcalcEnergyGradient( NewNBP *vNBP, int *num, double *result,
ThreeCoords *pregrad)
{
	__shared__ double res[THREADS_PER_BLOCK];
	int tid = threadIdx.x + blockIdx.x * blockDim.x;
	int buff_index= threadIdx.x;
	int i, num_interactions=*num;
	double electrostatic = 0.0, lennard_jones = 0.0;
	res[buff_index]=0;
	.
	.
	.

	if( tid<num_interactions )
	{
		.
		.//càlcul de l'energia (electrostatic i lennard_jones) i del
		.  gradient (dfdr i drdx/y/z)
		.
		//desem el resultat del gradient de la parella
		pregrad[ tid ].x= dfdr*drdx;
		pregrad[ tid ].y= dfdr*drdy;
		pregrad[ tid ].z= dfdr*drdz;

		//desem el resultat de l'energia de la parella en la memoria del block
		res[buff_index]=electrostatic+lennard_jones;

		//esperem que tots els threads hagin calculat l'energia i el gradient
		__syncthreads();

		//fem la reducció dels resultats de l'energia dels threads del block
```

```
            i=blockDim.x/2;
            while(i > 0){
                  if(buff_index < i){
                        res[buff_index]+=res[buff_index+i];
                  }
                  __syncthreads();
                  i/=2;
            }

            //guardem l'energia calculada de tots els threads del block
            if(buff_index==0)
                  result[blockIdx.x]=res[0];
      }
}
```

I.II.II CUDA iteració 2

Codi complet del kernel *kernelcalcEnergyGradient* utilitzant les noves estructures de dades:

```
__global__ void kernelcalcEnergyGradient( NewNBP *vNBP, NewAtom *vatom, double *coords,
int *num, double *result, ThreeCoords *pregrad)
{
      __shared__ double res[THREADS_PER_BLOCK];
      int tid = threadIdx.x + blockIdx.x * blockDim.x;
      int buff_index= threadIdx.x;
      int i, num_interactions=*num;
      double electrostatic = 0.0, lennard_jones = 0.0;
      res[buff_index]=0;

      double dx,dy,dz;
      double dist2,cgij;
      double sig2,epsij;
      double disti,dist2i;

      double rr2,rr6,rr12;

      double dfeldr, dfljdr, dfdr, drdx, drdy, drdz;

      // For each non-bonded interaction
      if( tid<num_interactions )
      {
            NewAtom *a = &vatom[vNBP[tid].atomA];
            NewAtom *b = &vatom[vNBP[tid].atomB];

            // Components of the vector which goes from one point to the other
            dx = coords[a->ix] - coords[b->ix];
            dy = coords[a->iy] - coords[b->iy];
            dz = coords[a->iz] - coords[b->iz];

            // Squared distance between points
            dist2=dx*dx+dy*dy+dz*dz;

            // Charge product
            cgij = vNBP[tid].cg;

            // Inverse of the distance and squared distance
            disti = 1.0/sqrt(dist2);
            dist2i = disti*disti;

            // Parameters for the Lennard-jones
            sig2 = vNBP[tid].sigma2;//4.0 * params_a->sigma *    params_b->sigma;
            epsij = a->epsilon * b->epsilon;

            // Coulomb
            electrostatic = cgij * disti;
```

```
            // Lennard-Jones
            rr2 = sig2*dist2i;
            rr6 = rr2*rr2*rr2;
            rr12 = rr6*rr6;
            lennard_jones = epsij * (rr12 - rr6);

            dfeldr = - cgij * dist2i;
            dfljdr = -6.0 * epsij * disti * (rr12 + rr12 - rr6);
            dfdr = dfeldr + dfljdr;
            drdx = dx*disti;
            drdy = dy*disti;
            drdz = dz*disti;

            // The gradient of the two atoms is updated
            // Gradient components are computed

            pregrad[ tid ].x= dfdr*drdx;
            pregrad[ tid ].y= dfdr*drdy;
            pregrad[ tid ].z= dfdr*drdz;

            res[buff_index]=electrostatic+lennard_jones;

            __syncthreads();

            //Reduction
            i=blockDim.x/2;
            while(i > 0){
                  if(buff_index < i){
                        res[buff_index]+=res[buff_index+i];
                  }
                  __syncthreads();
                  i/=2;
            }

            if(buff_index==0)
                  result[blockIdx.x]=res[0];
      }
}
```

Codi de *calcEnergyGradient* després de treure'n les etapes de pre-procés:

```
double CUDANonBondingTermVacuumCalculator::calcEnergyGradient(double * coords, double *
grad, vector< NonBondPair > & nonBondingInteractions)
{
      double *dev_result, *result, res=0;
      int numPairs = nonBondingInteractions.size();
      int num_blocks = ceil((double)numPairs  / (double) THREADS_PER_BLOCK);
      int *dev_numPairs, i;
      ThreeCoords *dev_tcoords, *tcoords;

      cudaMalloc( (void**)&dev_numPairs, sizeof(int));
      cudaMalloc( (void**)&dev_result, num_blocks*sizeof(double));
      cudaMalloc( (void**)&dev_tcoords, numPairs*sizeof(ThreeCoords) );

      cudaMemcpyAsync( dev_numPairs, &numPairs, sizeof(int), cudaMemcpyHostToDevice);

      kernelcalcEnergyGradient<<<num_blocks,THREADS_PER_BLOCK>>>(dev_vnbp,dev_vatom,
dev_coords, dev_numPairs, dev_result, dev_tcoords);

      tcoords=(ThreeCoords*)malloc(numPairs*sizeof(ThreeCoords));
      result=(double*)malloc(num_blocks*sizeof(double));

      cudaMemcpy(tcoords, dev_tcoords, numPairs*sizeof(ThreeCoords),
cudaMemcpyDeviceToHost);
      cudaMemcpyAsync(result, dev_result, num_blocks*sizeof(double),
```

```
cudaMemcpyDeviceToHost);
	for(i=0; i<numPairs;i++){
		grad[ nonBondingInteractions[i].first->ix ] += tcoords[i].x;
		grad[ nonBondingInteractions[i].first->iy ] += tcoords[i].y;
		grad[ nonBondingInteractions[i].first->iz ] += tcoords[i].z;
		grad[ nonBondingInteractions[i].second->ix ] -= tcoords[i].x;
		grad[ nonBondingInteractions[i].second->iy ] -= tcoords[i].y;
		grad[ nonBondingInteractions[i].second->iz ] -= tcoords[i].z;
	}
	cudaThreadSynchronize();
	for(i=0;i<num_blocks;i++){
		res+=result[i];
	}

	cudaFree(dev_result);
	cudaFree(dev_numPairs);
	cudaFree(dev_tcoords);
	free(result);
	free(tcoords);
	return res;
}
```

Codi de les funcions de pre-procés :

```
void CUDANonBondingTermVacuumCalculator::createAtomVector(){
	int i, size;

	size=vec_atoms.size();
	NewAtom *vatom= (NewAtom*)malloc(sizeof(NewAtom)*size);

	for(i=0;i<size;i++){
		if(vec_atoms[i]->nonBondingParams!=0)
			vatom[i].epsilon=vec_atoms[i]->nonBondingParams->epsilon;

		vatom[i].ix = vec_atoms[i]->ix;
		vatom[i].iy = vec_atoms[i]->iy;
		vatom[i].iz = vec_atoms[i]->iz;
	}

	cudaMalloc( (void**)&dev_vatom, size*sizeof(NewAtom) );
	cudaMemcpy( dev_vatom, vatom, size*sizeof(NewAtom), cudaMemcpyHostToDevice);

	free(vatom);
}

void CUDANonBondingTermVacuumCalculator::createNBIVector( vector< NonBondPair > &
nonBondingInteractions){
	int i, size;

	NewNBP *v=(NewNBP*)malloc(sizeof(NewNBP)*nonBondingInteractions.size());
	size=nonBondingInteractions.size();

	for(i=0;i<size;i++){
		v[i].atomA=nonBondingInteractions[i].first->vectorId;
		v[i].atomB=nonBondingInteractions[i].second->vectorId;
		v[i].cg=nonBondingInteractions[i].cg;
		v[i].sigma2=nonBondingInteractions[i].sigma2;
	}

	cudaMalloc( (void**)&dev_vnbp, size*sizeof(NewNBP) );
	cudaMemcpy( dev_vnbp, v, size*sizeof(NewNBP), cudaMemcpyHostToDevice);

	free(v);
}
```

```
void CUDANonBondingTermVacuumCalculator::copyCoordsToGPU(double *coords){
	cudaMalloc( (void**)&dev_coords, numCoords*sizeof(double));
	cudaMemcpyAsync( dev_coords, coords, numCoords*sizeof(double),
cudaMemcpyHostToDevice);
}
```

I.II.III CUDA iteració 3

Codi de *createMapVectors* encarregada de crear les dues taules de mapeig (crear les dues taules, ordenar-les i crear vectors d'inici):

```
void CUDANonBondingTermVacuumCalculator::createMapVectors(int numInteractions, int
numAtoms){

	int  *dev_size_atoms, *dev_size_nbp;
	int num_blocks = ceil((double)numInteractions  / (double) THREADS_PER_BLOCK);

	if(num_blocks>65534)
		num_blocks=65534;

	cudaMalloc((void **)&dev_map1k, numInteractions*sizeof(int));
	cudaMalloc((void **)&dev_map1v, numInteractions*sizeof(int));
	cudaMalloc((void **)&dev_map2k, numInteractions*sizeof(int));
	cudaMalloc((void **)&dev_map2v, numInteractions*sizeof(int));
	cudaMalloc((void **)&dev_start1, numAtoms*sizeof(int));
	cudaMalloc((void **)&dev_start2, numAtoms*sizeof(int));
	cudaMalloc((void **)&dev_size_atoms, sizeof(int));
	cudaMalloc((void **)&dev_size_nbp, sizeof(int));
	cudaMemcpy(dev_size_nbp, &numInteractions, sizeof(int), cudaMemcpyHostToDevice);
	cudaMemcpy(dev_size_atoms, &numAtoms, sizeof(int), cudaMemcpyHostToDevice);

	kernelCreateMaps<<< num_blocks, THREADS_PER_BLOCK >>>( dev_vnbp, dev_map1k,
dev_map1v, dev_map2k, dev_map2v, dev_size_nbp );

	thrust::device_ptr<int> dev_ptr1k(dev_map1k);
	thrust::device_ptr<int> dev_ptr1v(dev_map1v);
	thrust::device_ptr<int> dev_ptr2k(dev_map2k);
	thrust::device_ptr<int> dev_ptr2v(dev_map2v);

	thrust::sort_by_key( dev_ptr1k, dev_ptr1k + numInteractions, dev_ptr1v);
	thrust::sort_by_key( dev_ptr2k, dev_ptr2k + numInteractions, dev_ptr2v);

	num_blocks = ceil((double)numAtoms  / (double) THREADS_PER_BLOCK);
	if(num_blocks>65534)
		num_blocks=65534;
	createStart<<<num_blocks, THREADS_PER_BLOCK>>>(dev_map1k, dev_start1,
dev_size_atoms, dev_size_nbp);
	createStart<<<num_blocks, THREADS_PER_BLOCK>>>(dev_map2k, dev_start2,
dev_size_atoms, dev_size_nbp);

	cudaFree(dev_size_atoms);
	cudaFree(dev_size_nbp);

}
```

Codi de *kernelCreateMaps* , encarregat de crear una taula amb l'identificador de l'àtom i la posició de la INC a la que pertany:

```
__global__ void kernelCreateMaps(NewNBP *nbi, int *m1k,int *m1v,int *m2k, int *m2v, int
*sizeNBP ){
	int i = threadIdx.x + blockIdx.x * blockDim.x;

	while(i<*sizeNBP){
		m1k[i]=nbi[i].atomA;
```

```
            m1v[i]=i;
            m2k[i]=nbi[i].atomB;
            m2v[i]=i;

            i+=gridDim.x*blockDim.x;
      }
}
```

Codi de *kernelcreateStart* , encarregat de buscar i guardar la primera posició dels àtoms en la taula de mapeig:

```
__global__ void kernelcreateStart(int *m1k, int *start, int *sizeAtoms, int *sizeNbi){

      int tid = threadIdx.x + blockIdx.x * blockDim.x;
      int ld, le, i, b;

      while(tid<*sizeAtoms){
            b=0;
            le=0;
            ld=*sizeNbi-1;

            //cerca dicotomica per trobar l'atom al vector m1k
            while(le<=ld && !b){
                  i=(ld+le)/2;
                  if(m1k[i]==tid)
                        b=1;
                  else if(m1k[i]>tid)
                        ld=i-1;
                  else
                        le=i+1;
            }

            //si hem trobat l'atom busquem el seu primer element
            if(b){
                  while((i>=0) && (m1k[i]==tid)){
                        i--;
                  }
                  start[tid]=i+1;
            }
            else{
                  start[tid]=-1;
            }
            tid+=gridDim.x*blockDim.x;
      }
}
```

Codi de *kernelgradReduction* , encarregat de realitzar la reducció del gradient:

```
__global__ void kernelgradReduction(double *grad, NewAtom *va,ThreeCoords *tcoords, int
*map1k, int *map1v, int *map2k, int *map2v, int *start1,int *start2,int *numAtoms, int
*numPairs){
      int tid = threadIdx.x + blockIdx.x * blockDim.x;
      int i;
      double x, y, z;

      while(tid<*numAtoms){
            x=0;
            y=0;
            z=0;

            if(start1[tid]!=-1){
                  for(i=start1[tid];(i<*numPairs) && (map1k[i]==tid);i++){
                        x+=tcoords[map1v[i]].x;
                        y+=tcoords[map1v[i]].y;
                        z+=tcoords[map1v[i]].z;
```

```
                }
        }

        if(start2[tid]!=-1){
                for(i=start2[tid];(i<*numPairs) && (map2k[i]==tid);i++){
                        x-=tcoords[map2v[i]].x;
                        y-=tcoords[map2v[i]].y;
                        z-=tcoords[map2v[i]].z;
                }
        }
        grad[va[tid].ix]=x;
        grad[va[tid].iy]=y;
        grad[va[tid].iz]=z;

        tid+=gridDim.x*blockDim.x;
    }
}
```

Codi de *kernelgradReduction2* , versió modificada de *kernelgradReduction* minimitzant accessos a memòria global:

```
__global__ void kernelgradReduction2(double *grad, NewAtom *va,ThreeCoords *tcoords, int
*map1k, int *map1v, int *map2k, int *map2v, int *start1,int *start2,int *numAtoms, int
*numPairs){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int i,maxim, j;
    double x, y, z;

    while(tid<*numAtoms){
        x=0;
        y=0;
        z=0;
        if(start1[tid]!=-1){
            if(tid== *numAtoms-1){
                maxim=*numPairs;
            }else if(start1[tid+1]!=-1){
                maxim=start1[tid+1];
            }else{
                for(j=tid+2;j<*numAtoms && start1[j]==-1;j++);
                if(j<*numAtoms)
                    maxim=start1[j];
                else
                    maxim=*numPairs;
            }
            for(i=start1[tid];i<maxim;i++){
                x+=tcoords[map1v[i]].x;
                y+=tcoords[map1v[i]].y;
                z+=tcoords[map1v[i]].z;
            }
        }

        if(start2[tid]!=-1){
            if(tid== *numAtoms-1){
                maxim=*numPairs;
            }else if(start2[tid+1]!=-1){
                maxim=start2[tid+1];
            }else{
                for(j=tid+2;j<*numAtoms && start2[j]==-1;j++);
                if(j<*numAtoms)
                    maxim=start2[j];
                else
                    maxim=*numPairs;
            }
            for(i=start2[tid];i<maxim;i++){
                x-=tcoords[map2v[i]].x;
```

```
                        y-=tcoords[map2v[i]].y;
                        z-=tcoords[map2v[i]].z;
                  }
            }
            grad[va[tid].ix]=x;
            grad[va[tid].iy]=y;
            grad[va[tid].iz]=z;
            tid+=gridDim.x*blockDim.x;
      }
}
```

Codi de *calcEnergyGradient* realitzant la reducció del gradient a la GPU:

```
double CUDANonBondingTermVacuumCalculator::calcEnergyGradient(double * coords, double *
grad, vector< NonBondPair > & nonBondingInteractions)
{
      double *dev_result, *result, res=0;
      int numPairs = nonBondingInteractions.size();
      int num_blocks = ceil((double)numPairs  / (double) THREADS_PER_BLOCK);
      int *dev_numPairs, i;
      ThreeCoords *dev_tcoords;
      double *dev_grad;
      int *dev_size_atoms, size_atoms=vec_atoms.size();

      if(num_blocks>65534)
            num_blocks=65534;

      cudaMalloc( (void**)&dev_numPairs, sizeof(int));
      cudaMalloc( (void**)&dev_result, num_blocks*sizeof(double));
      cudaMalloc( (void**)&dev_tcoords, numPairs*sizeof(ThreeCoords) );

      cudaMemcpyAsync( dev_numPairs, &numPairs, sizeof(int), cudaMemcpyHostToDevice);

      kernelcalcEnergyGradient<<<num_blocks,THREADS_PER_BLOCK>>>(dev_vnbp,dev_vatom,
dev_coords, dev_numPairs, dev_result, dev_tcoords);

      int num_blocks2 = ceil((double)size_atoms / (double) THREADS_PER_BLOCK);
      if(num_blocks2>65534)
            num_blocks2=65534;
      cudaMalloc((void **)&dev_grad, size_atoms*3*sizeof(double));
      cudaMalloc((void **)&dev_size_atoms, sizeof(int));
      cudaMemcpy(dev_size_atoms, &size_atoms, sizeof(int), cudaMemcpyHostToDevice);

      kernelgradReduction2<<<num_blocks2, THREADS_PER_BLOCK>>>(dev_grad, dev_vatom,
dev_tcoords, dev_map1k, dev_map1v, dev_map2k, dev_map2v, dev_start1, dev_start2,
dev_size_atoms, dev_numPairs);

      result=(double*)malloc(num_blocks*sizeof(double));

      cudaMemcpy(result, dev_result, num_blocks*sizeof(double), cudaMemcpyDeviceToHost);
      cudaMemcpyAsync(grad, dev_grad, size_atoms*3*sizeof(double),
cudaMemcpyDeviceToHost);

      for(i=0;i<num_blocks;i++){
            res+=result[i];
      }

      cudaFree(dev_size_atoms);
      cudaFree(dev_grad);
      cudaFree(dev_result);
      cudaFree(dev_numPairs);
      cudaFree(dev_tcoords);
      free(result);
      free(tcoords);
      return res;
```

```
}
```

I.II.III OpenCL

Codi de la inicialització d'OpenCL:

```
void OpenCLNonBondingTermVacuumCalculator::initOpenCL(){

	context=0;
	commandQueue=0;
	device=0;
	program=0;

	context= CreateContext();
	if(context==0){
		cerr << "Failed to create OpenCL context"<<endl;
		exit (-1);
	}
	commandQueue = CreateCommandQueue(context, &device);
	if(commandQueue==0){
		cerr << "Failed to create OpenCL command-queue"<<endl;
		exit (-1);
	}
	program= CreateProgram(context, device, "kernels.cl");
	if(program==0){
		cerr << "Failed to create OpenCL program"<<endl;
		exit (-1);
	}
	cout<<"Init OpenCL finished"<<endl;
}

cl_context CreateContext(){
	cl_int errNum;
	cl_uint numPlatforms;
	cl_platform_id *platformsId;
	cl_context context=0;
	int i, b;

	errNum= clGetPlatformIDs(0, NULL, &numPlatforms);
	if(errNum != CL_SUCCESS || numPlatforms <=0){
		cerr << "Failed to find OpenCL platforms" <<endl;
		return 0;
	}
	platformsId= (cl_platform_id *)malloc(numPlatforms*sizeof(cl_platform_id));
	errNum= clGetPlatformIDs(numPlatforms, platformsId, NULL);
	if(errNum != CL_SUCCESS){
		cerr << "Error to find OpenCL platforms" <<endl;
		return 0;
	}

	b=0;
	for(i=0;i<numPlatforms && !b;i++){

		cl_context_properties contextProperties[] ={CL_CONTEXT_PLATFORM,
(cl_context_properties)platformsId[i],0};

		context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_GPU, 0,
0, &errNum);
		if(errNum == CL_SUCCESS)
			b=1;
	}
	if(errNum != CL_SUCCESS){
		cerr << "Failed to create OpenCL GPU context -> Error number: " << errNum
<<endl;
		return 0;
	}
```

```
	return context;
}

cl_command_queue CreateCommandQueue(cl_context context, cl_device_id *device){
	cl_int errNum;
	cl_device_id *devices;
	cl_command_queue commandQueue = 0;
	size_t deviceBufferSize= -1;

	errNum = clGetContextInfo (context, CL_CONTEXT_DEVICES, 0, 0, &deviceBufferSize);
	if(errNum != CL_SUCCESS){
		cerr<< "Failed call to clGetContextInfo()"<<endl;
		return 0;
	}

	if (deviceBufferSize <=0){
		cerr<<"No devices available";
		return 0;
	}

	devices = new cl_device_id[deviceBufferSize / sizeof(cl_device_id)];
	errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, deviceBufferSize, devices,
0);
	if(errNum != CL_SUCCESS){
		cerr<<"Failed to get device IDs";
		return 0;
	}

	commandQueue = clCreateCommandQueue(context, devices[0], 0, 0);

	if (commandQueue ==0){
		cerr <<"Failed to create commandQueue for device 0";
		return 0;
	}

	*device = devices[0];
	delete [] devices;
	return commandQueue;
}

cl_program CreateProgram(cl_context context, cl_device_id device, const char* fileName){
	cl_int errNum;
	cl_program program;

	ifstream kernelFile(fileName, ios::in);
	if (!kernelFile.is_open()){
		cerr << "Failed to open file for reading: " << fileName <<endl;
		return 0;
	}

	ostringstream oss;
	oss << kernelFile.rdbuf();
	string srcStdStr = oss.str();
	const char *srcStr = srcStdStr.c_str();
	program = clCreateProgramWithSource(context, 1,(const char**)&srcStr,0, 0);
	if (program == 0)   {
		cerr << "Failed to create CL program from source." << endl;
		return 0;
	}

	errNum = clBuildProgram(program, 0, 0, 0, 0, 0);
	if (errNum != CL_SUCCESS){
		// Determine the reason for the error
		char buildLog[16384];
		clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
sizeof(buildLog), buildLog, 0);
		cerr << "Error in kernel: " << endl;
```

```
            cerr << buildLog;
            clReleaseProgram(program);
            return 0;
      }
      return program;
}
```

Codi de *BitonicSort* , encarregada de preparar les dades i executar els kernels d'ordenació:

```
void BitonicSort(int size,cl_mem keys, cl_mem values)
{
      cl_kernel kernel_sort_s, kernel_sort_m;
      int i,j, r_size, *pad1, *pad2, padding; //exp=2^x
      cl_mem dev_keys, dev_bb, dev_rb, dev_values;
      cl_int errNum;

      for(r_size=1; r_size<size;r_size<<=1);

      padding =  r_size-size ;

      dev_rb = clCreateBuffer(context,CL_MEM_READ_WRITE, sizeof(int) ,NULL,NULL);
      dev_bb = clCreateBuffer(context,CL_MEM_READ_WRITE, sizeof(int) ,NULL,NULL);
      dev_keys = clCreateBuffer(context,CL_MEM_READ_WRITE , sizeof(int)*r_size
,NULL,NULL);
      dev_values = clCreateBuffer(context,CL_MEM_READ_WRITE , sizeof(int)*r_size
,NULL,NULL);

      clEnqueueCopyBuffer(commandQueue,keys, dev_keys, 0, 0, size*sizeof(int), 0, 0, 0);
      clEnqueueCopyBuffer(commandQueue, values, dev_values, 0, 0, size*sizeof(int), 0,
0, 0);
      if(padding>0){
            pad1= (int *)malloc(padding*sizeof(int));
            pad2= (int *)malloc(padding*sizeof(int));
            for(i=0;i<padding;i++){
                  pad1[i]=MAXINT;
                  pad2[i]=0;
            }
            clEnqueueWriteBuffer(commandQueue, dev_keys, CL_TRUE, size*sizeof(int),
padding*sizeof(int), pad1, 0, 0, 0);
            clEnqueueWriteBuffer(commandQueue, dev_values, CL_TRUE, size*sizeof(int),
padding*sizeof(int), pad2, 0, 0, 0);
      }

      kernel_sort_s = clCreateKernel(program, "bitonic_simple_v3", 0);
      if(kernel_sort_s==0){
            cerr << "Failed to create OpenCL kernel bitonic_s"<<endl;
            exit (-1);
      }

      kernel_sort_m = clCreateKernel(program, "bitonic_multi_v3", 0);
      if(kernel_sort_m==0){
            cerr << "Failed to create OpenCL kernel bitonic_m"<<endl;
            exit (-1);
      }

      errNum = clSetKernelArg(kernel_sort_s, 0, sizeof(cl_mem), &dev_bb);
      errNum |= clSetKernelArg(kernel_sort_s, 1, sizeof(cl_mem), &dev_rb);
      errNum |= clSetKernelArg(kernel_sort_s, 2, sizeof(cl_mem), &dev_keys);
      errNum |= clSetKernelArg(kernel_sort_s, 3, sizeof(cl_mem), &dev_values);
      errNum |= clSetKernelArg(kernel_sort_m, 0, sizeof(cl_mem), &dev_bb);
      errNum |= clSetKernelArg(kernel_sort_m, 1, sizeof(cl_mem), &dev_rb);
      errNum |= clSetKernelArg(kernel_sort_m, 2, sizeof(cl_mem), &dev_keys);
      errNum |= clSetKernelArg(kernel_sort_m, 3, sizeof(cl_mem), &dev_values);
      if(errNum != CL_SUCCESS){
            cerr<<"Error setting arguments for bitonic "<<errNum << endl;
```

```
            exit(-1);
      }

      size_t nb=r_size/2;
      size_t nt;
      if(NUM_THREADS_BITONIC > (r_size/2))
            nt=r_size;
      else
            nt=NUM_THREADS_BITONIC;

      for(i=1; i < r_size; i<<=1){
            for(j=i; j>(nt/2); j>>=1){
                  clEnqueueWriteBuffer(commandQueue, dev_bb, CL_TRUE, 0, sizeof(int),
&i, 0, 0, 0);
                  clEnqueueWriteBuffer(commandQueue, dev_rb, CL_TRUE, 0, sizeof(int),
&j, 0, 0, 0);
                  errNum = clEnqueueNDRangeKernel(commandQueue, kernel_sort_s, 1, 0,
&nb, &nt, 0, 0, 0);
                  if(errNum != CL_SUCCESS){
                        cerr<<"Error queuing kernel for execution"<<errNum<<endl;
                        exit(-1);
                        }
            }
            clEnqueueWriteBuffer(commandQueue, dev_bb, CL_TRUE, 0, sizeof(int), &i, 0,
0, 0);
            clEnqueueWriteBuffer(commandQueue, dev_rb, CL_TRUE, 0, sizeof(int), &j, 0,
0, 0);
            errNum = clEnqueueNDRangeKernel(commandQueue, kernel_sort_m, 1, 0, &nb,
&nt, 0, 0, 0);
            if(errNum != CL_SUCCESS){
                  cerr<<"Error queuing kernel for execution"<<errNum<<endl;
                  exit(-1);
            }
      }

      clEnqueueCopyBuffer(commandQueue,dev_keys, keys, 0, 0, size*sizeof(int),0,0,0);
      clEnqueueCopyBuffer(commandQueue,dev_values, values,0,0,size*sizeof(int),0,0,0);

      clReleaseMemObject(dev_keys);
      clReleaseMemObject(dev_values);
      clReleaseMemObject(dev_bb);
      clReleaseMemObject(dev_rb);
}
```

Codis dels kernels *bitonic_simple_v3* i *bitonic_multi_v3*, encarregats de fer els intercanvis d'element per ordenar un vector de claus i valors:

```
kernel void bitonic_simple_v3 (__global const int* bluebox, __global const int* redbox,
__global int* keys, __global int* values)
{
      int id = get_global_id(0);
      unsigned int j; //j=2^x
      bool dir, en, swap; //dir -> 0=descendent (0->1) 1=ascendent (2<-3)
      int k1, k2, v1, v2, pos;

      j=*redbox;

      en= ((id & j) == 0);
      if(!en)
            id+=(get_global_size(0))-i;

      dir= ((id & (*bluebox<<1)) != 0);
      pos = id ^i;
      k1=keys[id];
      k2=keys[pos];
      v1 = values[id];
```

```
	v2 = values[pos];
	barrier(CLK_GLOBAL_MEM_FENCE);
	swap= ((k1>k2) && (dir==0)) || ((k1<k2) && (dir==1));
	if(swap){
		keys[id]=k2;
		keys[pos]=k1;
		values[id]=v2;
		values[pos]=v1;
	}
}

kernel void bitonic_multi_v3(__global const int* bluebox, __global const int* redbox,
__global int* keys, __global int* values)
{
	int id = get_global_id(0);
	int id2=id;
	unsigned int j;
	bool dir, dir2, dir1, en, swap; //dir -> 0=descendent (0->1) 1=ascendent (2<-3)
	int k1, k2, v1, v2, pos;
	int gs=get_global_size(0);
	int  auxdir = *bluebox <<1;

	for(j=*redbox; j>0 ; j=j>>1){
		id=id2;
		en= ((id & j) == 0);
		if(!en)
			id+=gs-j;

		dir= ((id & auxdir) != 0);
		pos = id ^j;
		k1=keys[id];
		k2=keys[pos];
		v1 = values[id];
		v2 = values[pos];
		barrier(CLK_GLOBAL_MEM_FENCE);
		swap= ((k1>k2) && (dir==0)) || ((k1<k2) && (dir==1));
		if(swap){
			keys[id]=k2;
			keys[pos]=k1;
			values[id]=v2;
			values[pos]=v1;
		}
		barrier(CLK_GLOBAL_MEM_FENCE);
	}
}
```

I.II.IV Funcions per mesurar el temps

Codi exemple en l'ús de la funció *clock* :

```
clock_t start, end;
start=clock();
//codi a mesurar
end = clock();
cout << "Temps: " <<" Time all:"<<(double)(end-start)/1000000 <<endl;
```

Codi de les macros TIMER:

```
#include <sys/timeb.h>

	#define TIMER_CREATE(N) timespec N##_time;\
						unsigned long int N##_time_start,
N##_time_end;\
						unsigned long int N##_global;\
```

```
                                                unsigned int N##_times;\
                                                N##_global=0;\
                                                N##_times = 0;

      #define TIMER_MEASURE(N, C) clock_gettime(CLOCK_REALTIME, &N##_time); \
                                                N##_time_start = (unsigned long
int)N##_time.tv_sec * 1000LL + (unsigned long int)N##_time.tv_nsec / 1000000LL;\
                                                C \
                                                clock_gettime(CLOCK_REALTIME,
&N##_time); \
                                                N##_time_end = (unsigned long
int)N##_time.tv_sec * 1000LL + (unsigned long int)N##_time.tv_nsec / 1000000LL; \
                                                N##_global += (N##_time_end -
N##_time_start); \
                                                N##_times++;

      #define TIMER_REPORT(N) cout<<#N << " took " << ((double)N##_global ) / 1000 <<"
seconds ("<<(((double)N##_global)/N##_times) /1000<<"s mean exec. time)"<< endl;
```

Ús de les macros TIMER:

```
TIMER_CREATE(NBIVector)
TIMER_MEASURE(NBIVector, funcio_a_mesurar(...); )
TIMER_REPORT(NBIVector)
```

Exemple en l'ús d'esdeveniments CUDA per mesurar el temps d'execució dels kernels:

```
cudaEvent_t  cstart, cstop;
cudaEventCreate( &cstart ) ;
cudaEventCreate( &cstop );
cudaEventRecord( cstart, 0 );

kernel_a_mesurar<<<X,Y>>>(...)

cudaEventRecord( cstop, 0 );
cudaEventSynchronize( cstop );
float elapsedTime;
cudaEventElapsedTime( &elapsedTime,cstart, cstop );
printf( "Time to generate:      %3.1f ms\n", elapsedTime );
cudaEventDestroy( cstart );
cudaEventDestroy( cstop );
```

I.III Versions bitonic sort

Les altres dues versions implementades de l'algorisme Bitonic sort, que tenen un rendiment inferior, són molt similars a la versió explicada en l'apartat 6.3.2, però amb les següents diferencies:

- Versió 1: S'assigna un thread a cada element a ordenar. Com que cada comparació té dos elements i per tant dos threads, el swap només el realitzarà el primer element. Això fa que només s'utilitzin la meitat de threads però es molt senzill d'implementar. Els noms dels kernels són *bitonic_simple_v1* i *bitonic_multi_v1*.
- Versió 2: És una millora de la versió 1 on s'utilitza la shared memory per minimitzar els accesos a memòria global en el bitonic_sort_multi. Els noms dels kernels són *bitonic_simple_v2* i *bitonic_multi_v2*.

I.IV Fitxers adjunts

El fitxer comprimit adjunt a la documentació del projecte, conté el codi font del test amb les funcions paral·lelitzades i el callgraph, en format visual, dels profiles obtinguts per a tres jocs de proves. Degut a la mida de les imatges del profile no s'han pogut inserir en el document i per això es subministren en format digital.

El contingut del fitxer és el següent:

- Codi del test (codis\test) amb les implementacions finals de les funcions en CUDA (codis\test\Energy\CUDA) i OpenCL (codis\test\Energy\OpenCL).
- Codi de l'adaptació OpenCL per a la targeta AMD (codis\adaptacio_AMD).
- Codi de la funció modificada *calcEnergyGradient* (amb un loop unrolling de grau 2) utilitzant OmpSs (codis\OmpSs).
- Imatges del callgraph (profile).

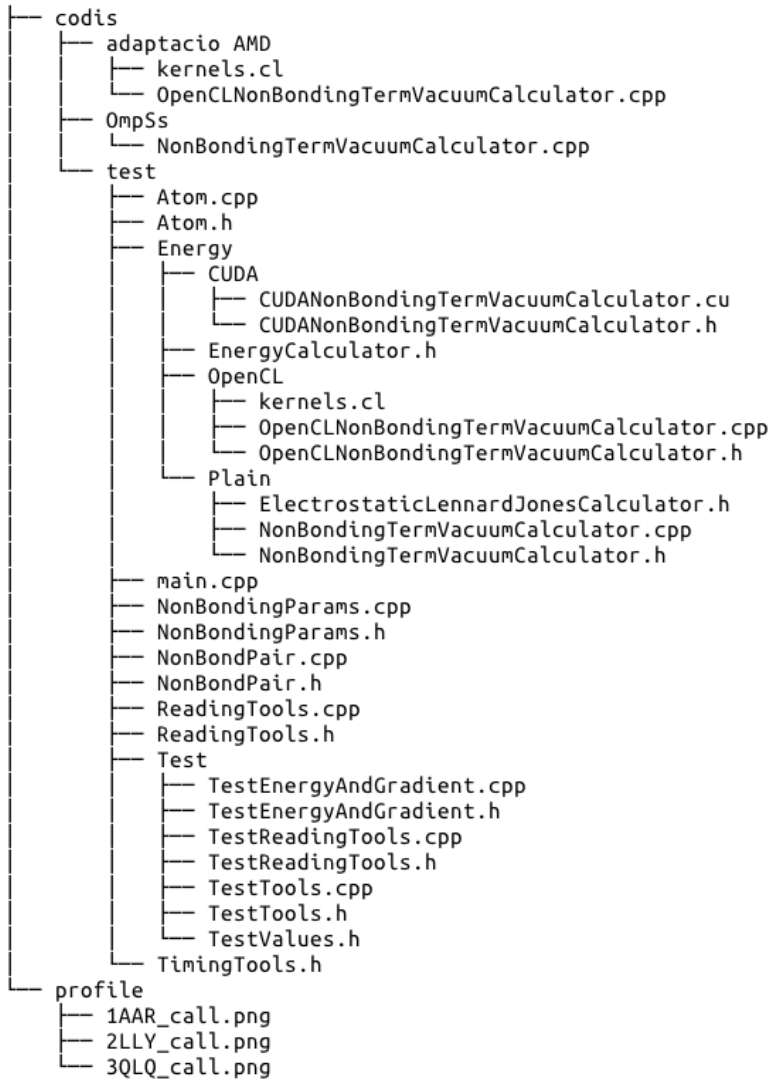


Fig. 42 Arbre de directoris i fitxers adjunts.